From: AAAI-92 Proceedings. Copyright ©1992, AAAI (www.aaai.org). All rights reserved.

Achieving the functionality of filter conditions in a partial order planner

Gregg Collins and Louise Pryor
The Institute for the Learning Sciences
Northwestern University
1890 Maple Avenue, Evanston IL 60201
collins@ils.nwu.edu, pryor@ils.nwu.edu

Abstract

One of the most common modifications made to the standard STRIPS action representation is the inclusion of filter conditions. A key function of such filter conditions is to distinguish between operators that represent different context-dependent effects for the same action. We consider how filter conditions may be used to provide this functionality in a complete and correct partial order planner. We conclude that they are not effective, and that in general the use of filter conditions is incompatible with the basic assumptions that lie behind partial order planning. We present an alternative mechanism, using the *secondary preconditions* of Pednault (1988, 1991) to represent context-dependent effects. The use of secondary preconditions is effective, and preserves completeness and correctness.

1 Introduction

Planning algorithms that are designed to possess desirable theoretical properties tend to ignore many of the functional considerations faced by practical planners. In particular, planners whose purpose is to demonstrate provably complete and correct algorithms (Chapman 1987, McAllester & Rosenblitt 1991), use a highly restricted version of the standard STRIPS[1] action representation (Fikes & Nilsson 1971). In contrast, many planners that are intended for practical use introduce extensions to the basic representation, notably *filter conditions* (Tate 1977, Wilkins 1988, Currie & Tate 1991). In this paper we investigate the effects of adding filter conditions to a provably complete and correct planner, and present an alternative mechanism that provides much of the same functionality in a cleaner and more efficient way.

1.1 Filter conditions

A *filter condition* for an action is a precondition that does not become a new subgoal. Typical uses of filter conditions include the following:

- In the Charniak & McDermott (1985) version of the blocks world, the operator to move an object has the filter condition *the object is a block.*
- In SIPE (Wilkins 1988), the operator that fetches objects from room1 has the filter condition *the object is in room1.*
- In O-Plan (Currie & Tate 1991), the operator for a particular method of oil rig construction has the filter condition *the soil type is sandy.*

In general, filter conditions are used to judge the applicability of an operator. If an operator has filter conditions that are not already true in the current partial plan the use of the operator is ruled out. Filter conditions are thus treated very differently from ordinary preconditions, which become new subgoals to be achieved through subsequent planning. In this paper we consider the use of filter conditions in accounting for context-dependent effects, although they have been used for several other purposes as well (Collins & Pryor 1992).

Filter conditions account for context-dependent effects by distinguishing between otherwise similar action representations so that the correct effects can be represented. For example, in the blocks world the operator for picking up a block that is supported by another block clears that block, while the operator for picking up a block that is supported by the table has no equivalent effect (since the table is always considered to be clear). In order to ensure that the correct effects are asserted, Charniak & McDermott (1985) define two pickup operators, one with the filter condition (on ?X table), and the other with the filter condition (on ?X ?other) where ?other is constrained to be another block.

We have investigated the implementation of filter conditions in a complete and correct partial order planner. Our conclusion is that any such implementation fails to achieve the intended functionality of filter conditions. However, another extension to the basic STRIPS representation, *secondary preconditions* (Pednault 1988, 1991), can be integrated into the planner in a natural way, and provides the desired functionality in respect of context-dependent effects. We have implemented secondary preconditions in the partial order planner, and present a compari-

[1] In the basic STRIPS representation, an action consists of a name, a list of *preconditions*, a list of states that will be true after the successful execution of the action (the *add list)* and a list of states that will cease to be true after the successful execution of the action (the *delete list*).

A set of **steps**	The actions to be executed.
A set of **open conditions**	States to be achieved.
A set of **links**	Each link is of the form (*step1 state step2*), where *state* is required for the execution of *step2* and is achieved by *step1*.
A **partial ordering**	Ordering constraints on the steps.
A list of **unsafe links**	Those links for which there is a step (the **clobbering** step) that could potentially (according to the partial ordering) be executed between *step1* and *step2* and that would unachieve *state*.
A set of **codesignation constraints**	Required or forbidden bindings for the variables in the partial plan.

Figure 1: The structure of partial plans

son of the efficiency of filter conditions and secondary preconditions in accounting for context-dependent effects.

1.2 The basic algorithm

Our investigations were carried out using the systematic non-linear planner (SNLP), which was written by Barrett, Soderland & Weld (1991) using the algorithm of McAllester & Rosenblitt (1991). The basic action representation in SNLP is the usual STRIPS representation with preconditions, add list and delete list, augmented by codesignation constraints on the variables (Chapman 1987).

SNLP operates by searching the space of partial plans (figure 1). The basic algorithm is:

- The most promising partial plan is chosen from the search queue.
- If the partial plan is complete the search process is terminated and the resulting plan is returned.
- If there are any unsafe links, one such link is removed from the unsafe list and each modification shown in figure 2 is attempted. Each successful modification produces a new partial plan, which is added to the search queue. Links made unsafe due to the modification are added to the unsafe list.
- Otherwise, there is at least one open precondition, for a step that is therefore not yet enabled. One such precondition is removed from the open list and each modification shown in figure 3 is attempted. Each successful modification produces a new partial plan, which is added to the search queue. Links made unsafe due to the modification are added to the unsafe list.

Promotion	Introduce an ordering to ensure that the clobbering step occurs after the step at the end of the unsafe link.
Demotion	Introduce an ordering to ensure that the clobbering step occurs before the step at the beginning of the unsafe link.
Separation	Introduce codesignation constraints to ensure that the state resulting from the clobbering step cannot unify with the state in the unsafe link.

Figure 2: Modifications for unsafe links

Add new **step**	Find an operator with a proposition in its add list that can be unified with the open condition. Make the operator the new step, add its preconditions to the list of open conditions, and add its codesignation constraints. Add a link from the new step to the unenabled step.
Add new **link**	Find a step with a proposition in its add list that can be unified with the open condition. Add a link from the found step to the unenabled step.
Add link	Add the bindings necessary for the unification to the set of codesignation constraints. Make the appropriate ordering constraint.

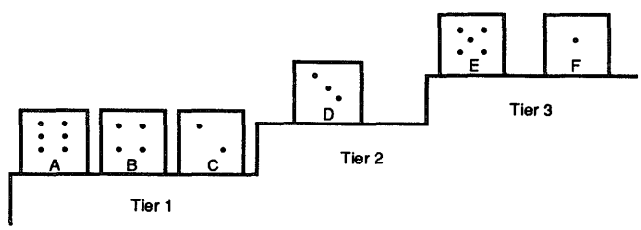
Figure 3: Modifications for open preconditions

1.3 Performance evaluation

We tested the planning algorithms described in this paper on an artificial domain designed to highlight the effects of operator selection on planning efficiency. The domain consists of a series of tiers, each of which can hold an infinite number of blocks. Each block may be in any of six orientations (like dice) (figure 4). There is just one type of action:

Raise moves a block up a tier while changing its orientation by a quarter turn to the next position in a cycle of the six.

In order to move a block from a tier, there must be another block on the same tier (no tier may be left empty as the result of a move). Goals are expressed as a required position, for example (on A tier3). In this domain loops are impossible, and there are no unachievable subgoals or expensive actions. Figure 5 shows two of the operators in this do-



In order to move block D to Tier 3, one of A,B,C must be moved first. After D has been moved, it will have face 4 uppermost.

Figure 4: Our test domain.

Raise a block		
Action:	(raise ?block)	(raise ?block)
Preconditions:	(on ?block tier1) (on ?other tier1)	(on ?block tier1) (on ?other tier1)
Filter conditions:	(up ?block face1)	(up ?block face6)
Add list:	(on ?block tier2) (up ?block face2)	(on ?block tier2) (up ?block face1)
Delete list:	(on ?block tier1) (up ?block face1)	(on ?block tier1) (up ?block face6)
Codesignation constraints	(not (?block ?other))	(not (?block ?other))

Figure 5: Operators for raising a block using filter conditions

main expressed in terms of a STRIPS representation modified to include filter conditions.

2 Implementing filters in SNLP

To test the feasibility of a straightforward implementation of filter conditions, we constructed a modified version of the SNLP planner using an extended STRIPS representation as shown in figure 5, above. The modified algorithm proceeds in exactly the same way as the standard SNLP algorithm until all outstanding subgoals of a partial plan are satisfied. At this point, the planner attempts to show that the existing plan satisfies all the filter conditions of its operators; if any filter condition is unsatisfied, the plan is rejected. In order to ensure that a filter condition is met, the planner may add constraints to the plan in the same way that it might to guarantee that unsafe links are not clobbered[2].

In the algorithm sketched above, filter conditions do not do much filtering. Because they are not considered until the plan is otherwise complete, filter conditions play no role in pruning the search space during plan construction. An important part of the functionality of filter conditions—ruling out unreasonable uses of individual operators in partial plans—is thus missing.

Clearly, we would prefer a planning algorithm in which filter conditions are considered earlier in the process. Unfortunately, this is problematic in SNLP, as, in fact, it would be in any partial order planner[3]. As long as a plan in such a planner remains incomplete, it cannot be stated with certainty that a particular filter condition in that plan cannot be established, since it is always possible that a step that establishes the condition will subsequently be added to the plan. It is therefore impossible to rule out any incomplete partial plan based on its filter conditions in a partial order planner. We can thus predict that a partial order planner using filter conditions will perform inefficiently. This hypothesis is supported by the empirical results we describe in section 4 below.

One possible compromise solution might be to prioritize the search through the plan space based on an estimate of the likelihood that the filter conditions in a given partial plan will ultimately be established. This method has the drawback that the information necessary to make such an estimate is not readily available. However, by counting the number of currently unestablished filter conditions a partial plan contains, the planner can make a crude guess as to the probability that the plan will eventually achieve these conditions. This information could then be used as one factor in the evaluation metric used to determine the order of search. To test this alternative mechanism for implementing filter conditions, we implemented a version of SNLP in which partial plans that include unestablished filter conditions are penalized in the search process. As the empirical results in section 4 show, this implementation performs better than the basic implementation of filter conditions, despite the extra work involved in checking the filter conditions after each modification.

Our implementations demonstrate that it is possible to implement filter conditions directly in SNLP. Unfortunately, as we have just seen, there are severe drawbacks to any such implementation, stemming not from any peculiarities of the SNLP algorithm but from the nature of partial order planning itself. These drawbacks led us to explore the alternative approach described below.

3 Secondary preconditions

Pednault (1988, 1991) presents a method that accounts for context-dependent effects directly, without the need for separate operators to distinguish representations of the same action. In his method, a distinction is drawn between those preconditions that are necessary in order for it to be possible to execute an action, which are termed *feasibility* or *primary* preconditions, and those preconditions that specify the effect of an action on a particular state, termed *secondary* preconditions. There are two types of secondary precondition:

- A *causation* precondition is a condition that must be true in order for a particular state to become true as a result of a particular action.
- A *preservation* precondition is a condition that must be true in order for a particular state to remain true through the execution of a particular action.

[2]Our implementation uses the normal mechanism for adding links to test the satisfiability of the filter conditions, and can thus be viewed as an adaptation of the technique suggested by McAllester & Rosenblitt (1991) for dealing with abstractions.
[3]Or in a total order planner that may insert a new step anywhere in the plan, such as that used by McDermott (1989).

Raise a block		
Action:	(raise ?block)	
Preconditions:	(on ?block ?t) (on ?other ?t)	
Add list:	(on ?block tier2)	(up ?block face1)
Causation preconditions:	(?t tier1)	(up ?block face6)
Delete list:	(on ?block tier1)	(up ?block face1)
Preservation preconditions:	(not (?t tier1))	(up ?block ?face) (not (?face?face6))
Codesignation constraints:	(not (?block ?other))	

Figure 6: Part of the raise operator using secondary preconditions

For instance, in our example domain, filter conditions can be used to distinguish between the various operators that raise a block, depending on the orientation and position of the block the planner wishes to raise. There are six turning operators, one for each of the six possible orientations. The operators for raising a block from different levels must also be distinguished. Using filter conditions, then, we need twelve operators to represent the single action of raising a block in our domain. Using secondary preconditions we need only a single representation of the raise action, part of which is shown in figure 6. The add list of this operator contains all conditions that may potentially be added by the action, each associated with its causation preconditions. If the causation preconditions cannot be established, the condition is not added by the action. Similarly, the delete list contains all conditions that may potentially be deleted by the action, each associated with its preservation preconditions. If the preservation preconditions can be established, then the condition will not be deleted.

3.1 Secondary preconditions in SNLP

Figure 6 shows part of the representation of the raise operator in our example domain using secondary preconditions.

Prevention	*If the clobbering arises through the [addition, deletion] of a desired state, prevent if by adding the [causation, preservation] preconditions of the relevant proposition on the [add, delete] list (required to be [false, true]) to the list of open conditions, and adding the [negation of the causation, preservation] codesignation constraints to the set of codesignation constraints.*

Figure 7: New modification for unsafe links *with secondary preconditions*

Add new **step**	*If the open condition is required to be false,* find an operator with a proposition in its *delete list* that can be unified with the open condition. Make the operator the new step, add its preconditions (required to be true) and its codesignation constraints. Add a new link from the new step to the unenabled step.
Add new **link**	*If the open condition is required to be false,* find a step with a proposition in its *delete list* that can be unified with the open condition. Add a link from the found step to the unenabled step.
Add link	*Ensure that the proposition is deleted by adding one of the preservation preconditions of the proposition on the delete list (required to be false) to the list of open conditions, or adding the negation of one of the preservation codesignation constraints of the proposition on the delete list.* Add the bindings necessary for the unification to the set of codesignation constraints. Make the appropriate ordering constraint.

Figure 8: New modifications for open preconditions *required to be false*

Each effect on the add list has causation preconditions and codesignation constraints associated with it, and similarly each effect on the delete list has associated preservation preconditions and codesignation constraints. These extra conditions are attached only if they are not already feasibility preconditions.

The basic SNLP algorithm can accommodate the addition of secondary preconditions without major modification[4]. The required changes concern the structure of open conditions, and the generation of possible modifications to a partial plan that may be made in order to resolve an unsafe link (see figure 7) or establish an open condition (see figure 8). An open condition may now be required to be either true or false. The truth of a condition is established, as before, by a link from an add condition of a prior step: the falsity of a condition is established analogously by a link from a delete condition of a prior step.

A condition may be required to be true if it is:

- An enabling precondition of a step.
- A causation precondition of an effect possibly added by a step when the addition is required in order to establish the truth of a further effect.
- A preservation precondition of an effect possibly deleted by a step when the deletion must be prevented so as not to clobber another condition.

A condition may be required to be false if it is:

- A causation precondition of an effect possibly added by a step when the addition must be prevented so as not to clobber another condition

[4]McDermott (1989) has implemented secondary preconditions in a total order planner. He does not compare their effectiveness with that of filter conditions.

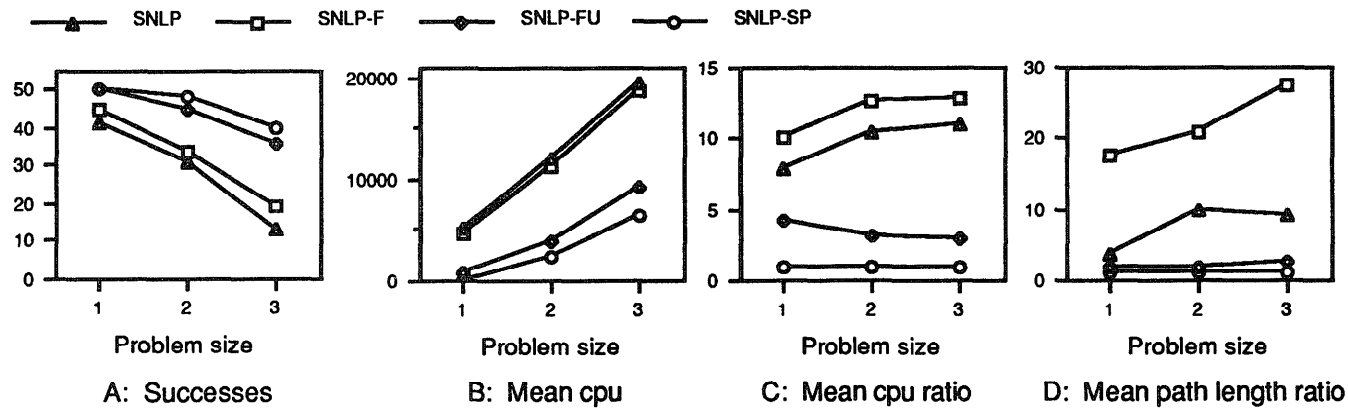


Figure 9: Performance of SNLP using secondary preconditions

- A preservation precondition of an effect possibly deleted by a step when the deletion is required in order to establish the falsity of a further effect

The secondary preconditions of an effect are never expanded into subgoals unless the effect either is used to establish a condition or must be prevented from clobbering an established condition. If neither of these cases applies, the truth or falsity of the effect has no influence on any other step in the plan, and it is ignored.

From this description of the modified algorithm it can be seen that secondary preconditions fit straightforwardly into the structure of SNLP, providing a neat way of representing context-dependent effects without multiplying the number of operators that are needed to represent the actions in a domain. Operators in this framework make the nature of context-dependent effects more explicit than do STRIPS operators with filter conditions. The modified algorithm preserves completeness and correctness.

4 Empirical results

The performance of a planner can be measured by either the efficiency of the plans that it produces, or the efficiency of its search procedure. However, in our example domain, it turns out that in all cases in which a plan was found it was the most efficient one possible. We have therefore evaluated the algorithms only in terms of their search efficiency.

We used three data sets for our evaluation, with one, two, or three blocks in the goal conditions. Each set consists of fifty randomly generated solvable problems. We ran four algorithms on each set, SNLP modified to include secondary preconditions (SNLP-SP) (see section 3), SNLP modified to include filter conditions (SNLP-F), SNLP modified to include filter conditions and using unestablished filter conditions to penalize partial plans in the search (SNLP-FU) (see section 2), and an unmodified version of SNLP.

Figure 9 shows a comparison of the results of running the four algorithms. Graph A shows the number of problems in each data set solved by each algorithm within a limit of 25000 units on the cpu time spent. SNLP-SP was the most successful algorithm: in no case did another algorithm solve a problem on which SNLP-SP failed. Graph B shows the mean cpu time taken by the four algorithms, with problems that were not solved taken into account at the time limit. Graph C presents the relative efficiency of the algorithms on those problems that they solve. For each problem, the time taken by each algorithm was divided by the time taken by SNLP-SP, and the graph shows the means of these ratios. Only those problems solved by the algorithm under consideration are included. Graph D presents a similar analysis of the number of nodes on the search path that led to a solution.

It can be seen that the algorithm using secondary preconditions, SNLP-SP, generally solved more problems within the time limit than the others, took less cpu to do so, and traversed a shorter search path. Moreover, our claim that filter conditions are not effective in a partial order planner because they cannot be used effectively to rule out unpromising plans is supported by the fact that SNLP-F performs very little better than the basic algorithm. We performed pairwise T-tests on planned comparisons between SNLP and SNLP-F, SNLP-F and SNLP-FU, and SNLP-FU and SNLP-SP for the number of successes as shown in graph A, the cpu time as shown in graph B, and the path length. For the latter two measures, we used the value at failure for those problems that were not solved within the time limit. The results for all problems taken together are shown in figure 10, and can be summarized as follows, at a 95% significance level:

Success rate: All three comparisons were significant, with the second algorithm in each pair solving more problems within the time limit.

Comparison	Successes		Cpu		Path length	
	Mean	T value	Mean	T value	Mean	T value
SNLP - SNLP-F	-0.087	-2.906	670	1.905*	-224.7	-4.903
SNLP-F - SNLP-FU	-0.220	-6.483	6943	8.886	396.4	7.122
SNLP-FU - SNLP-SP	-0.047	-2.701	1611	5.407	30.7	6.622

Figure 10: Mean differences and T-values ($p<0.05$ except *)

Cpu time: There was no significant difference between SNLP and SNLP-F. The other two comparisons were significant, with the second algorithm in each pair performing better.

Path length: SNLP-SP performed significantly better than SNLP-FU, which performed significantly better than SNLP-F. The other comparison was also significant, but in the reverse direction: SNLP had a shorter path length to solution than SNLP-F.

The results are analyzed further in (Collins & Pryor 1992).

An important difference between the algorithms that is not shown above is that the mean number of offspring of a partial plan, or branching factor, was much lower for SNLP-SP than for the other algorithms (1.7 compared to between 3.4 and 4.3). This implies that the efficiency advantage of using secondary preconditions will become progressively more pronounced as problems become more complex.

6 Conclusions

We have investigated the feasibility of implementing filter conditions in a provably complete and correct partial order planner. While such an implementation is possible, it must fail to achieve the desired functionality of filter conditions for reasons that stem from the fundamental assumptions behind partial order planning. We therefore sought an alternative mechanism through which to represent context-dependent effects, and found a candidate in the secondary preconditions of Pednault (1988, 1991). The implementation of secondary preconditions in a partial order planner is straightforward, and preserves completeness and correctness (Collins & Pryor 1992). A comparison of the empirical results obtained from the modified algorithms and the original algorithm supports our claim that the use of secondary preconditions accounts more efficiently for context-dependent effects than does the use of filter conditions.

Acknowledgments. Thanks to Brian Drabble, Eric Jones, Bruce Krulwich, Austin Tate and Dan Weld for useful discussions, to Dan Weld for supplying the original SNLP code, and to the reviewers for their comments. This work was supported in part by the AFOSR under grant number AFOSR-91-0341-DEF, and by DARPA, monitored by the ONR under contract N-00014-91-J-4092. The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting, part of The Arthur Andersen Worldwide Organization. The Institute receives additional support from Ameritech, an Institute Partner, and from IBM.

References

Allen, J., Hendler, J., & Tate, A. eds. 1990. *Readings in Planning*. San Mateo, CA: Morgan Kaufman.

Barrett, A., Soderland, S., & Weld, D. S. 1991. Effect of Step-Order Representations on Planning. Technical Report 91-05-06, Department of Computer Science and Engineering, University of Washington, Seattle.

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence*, 32: 333-337, and in (Allen *et al.* 1990).

Charniak, E., & McDermott, D. 1985. *Introduction to Artificial Intelligence*. Reading, MA: Addison-Wesley.

Collins, G., and Pryor, L. 1992. Representation and Performance in a Partial Order Planner. ILS tech report, in preparation.

Currie, K., & Tate, A. 1991. O-Plan: the open planning architecture. *Artificial Intelligence* 52: 49-86.

Fikes, R. E., & Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2: 189-208, and in (Allen *et al.* 1990).

McAllester, D., & Rosenblitt, D. 1991. Systematic Nonlinear Planning. Proceedings of the Ninth National Conference on Artificial Intelligence, Anaheim, CA.

McDermott, D. 1989. Regression Planning. Technical Report YALEU/CSD/RR #752, Computer Science Department, Yale University.

Pednault, E. P. D. 1988. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence* 4: 356-372

Pednault, E. P. D. 1991. Generalizing Nonlinear Planning to Handle Complex Goals and Actions with Context-Dependent Effects. Proceedings of the Twelfth International Conference on Artificial Intelligence, Darling Harbour, Sydney, Australia.

Tate, A. 1977. Generating project networks. IJCAI 1977, and in (Allen *et al.* 1990).

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufman