Generalizing the Order of Operators in Macro-Operators

RAYMOND J. MOONEY (MOONEY@SALLY.UTEXAS.EDU)

Department of Computer Sciences, University of Texas, Austin, TX 78712

Abstract

A number of machine learning systems have been built which learn *macro-operators* or *plan schemata*, i.e. general compositions of actions which achieve a goal. However, previous research has not addressed the issue of generalizing the temporal order of operators and learning macro-operators with partially-ordered actions. This paper presents an algorithm for learning partially-ordered macro-operators which has been incorporated into the EGGS domain-independent explanation-based learning system. Examples from the domains of computer programming and narrative understanding are used to illustrate the performance of this system. These examples demonstrate that generalizing the order of operators can result in more general as well as more justified concepts. A theoretical analysis of the time complexity of the generalization algorithm is also presented.

1. Introduction

A number of machine learning systems have been constructed which learn *macro-operators* or *plan schemata*, i.e. general compositions of actions which achieve a goal. The learning of macro-operators has been applied to a variety of domains including robot planning (Fikes, Hart, & Nilsson, 1972), puzzle solving (Korf, 1985; Laird, Rosenbloom, & Newell, 1986), and natural language text understanding (Mooney & DeJong, 1985). Recent explanation-based learning systems which operate in plan-based domains have addressed a variety of issues involving the learning of macro-operators. These issues include determining which macro-operators to learn (Minton, 1985), determining the appropriate level of generality for macro-operators (Segre, 1987), and learning iterative macro-operators (Shavlik & DeJong, 1987). However, an issue which has not been addressed in previous research is generalizing the temporal order of operators in a plan and thereby learning macro-operators which have partially-ordered actions. Previous macro-operator learning systems maintained the ordering of operators present in a specific training example and consequently could only acquire plans represented as linear sequences of actions.

This paper describes a procedure which generalizes the order of operators in plans composed of STRIPS operators (Fikes *et al.* 1972) and thereby learns macro-operators with partially-ordered actions. This procedure has been implemented as a part of the EGGS domain-independent explanation-based learning system (Mooney & Bennett, 1986). In addition to the normal process of explanation generalization, EGGS is now capable of computing the most general partial ordering of the actions in the plan within the constraint that the *structure* of the original explanation be maintained. In a plan-based domain, the structure of an explanation refers to the manner in which the effects of actions fulfill preconditions of subsequent actions. Generalizing the order of actions frequently results in a more general macro-operator which can be used to solve a larger class of problems.

2. An Example

A simple example which illustrates the advantage of learning macro-operators with partially ordered actions involves assignment statements in a programming language. In the domain of programming for parallel machines, knowing the most general partial ordering of a set of operators which achieves a goal is very important. Operators which do not have an ordering imposed upon them can be executed in parallel and knowing the most general partial ordering allows one to obtain the maximum amount of parallelism.

Given the STRIPS operator for an assignment statement shown in Table 1, consider the problem of achieving the goal: Value(A,2) $\wedge$ Value(C,4) given the initial state: Value(A,1), Value(B,2), Value(C,3), and Value(D,4). Assume that the plan of sequentially executing the operators Setq(A,B) and Setq(C,D) is either observed or generated to solve this specific problem. Figure 1 shows the explanation for how this plan achieves the specified goal. Performing standard explanation generalization which maintains unifications between effects and preconditions (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986) results in the generalized explanation shown in Figure 2.

Table 1: Variable Assignment Operator

Action	Preconditions	Effects

Setq(?a,?b)	Value(?a,?x)
	Value(?b,?y)
		Value(?a,?y)
		$\neg$Value(?a,?x)

In most cases, the two assignment operators in the generalized plan (i.e. Setq(?a1,?b1) and Setq(?a2,?b2)) can be executed in parallel and should not have an ordering constraint imposed upon them. For example, in the original problem, the two assignments Setq(A,B) and Setq(C,D) could be executed simultaneously. However, consider achieving the goal: Value(A,3) $\wedge$ Value(C,4) given the same initial state. The operator sequence: Setq(A,C), Setq(C,D) will achieve this goal; however, in this case, the sequential ordering of the operators is crucial. The operator Setq(C,D) must be executed after Setq(A,C) or else the value of C will be reset before it is referenced. In the planning literature, this problem is generally referred to as a *protection violation* since the state: Value(C,3) must be protected until it is needed as a precondition for Setq(A,C). Whenever ?b1=?a2 or ?b2=?a1 in an instantiation of the "double Setq" macro-operator, an ordering constraint must be imposed on the two assignment statements in order to prevent a protection violation. As this example illustrates, determining the conditions under which protection violations will occur and imposing appropriate ordering constraints to prevent them is the crucial step in learning a partially ordered macro-operator.

3. Overview of EGGS

This section presents a brief overview of the EGGS domain-independent explanation-based learning system which underlies the learning of partially ordered macro-operators. Mooney (1988a) presents a complete description of the EGGS system.

EGGS is a domain-independent explanation-based learning system which is capable of generalizing explanations of the following types: (1) proofs composed of Horn-clauses; (2) rewrite traces composed of term rewriting rules; and (3) plans composed of STRIPS operators. For each of these representation languages, EGGS has a module for building explanations by either solving problems itself or by explaining (verifying) problem solutions

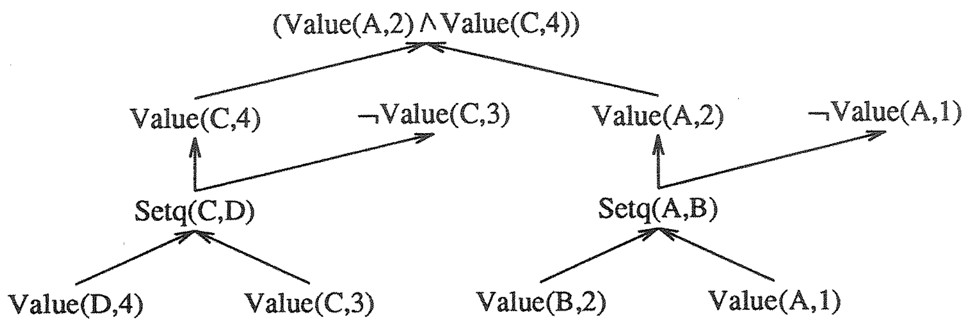

Figure 1: Variable Assignment Example -- Specific Explanation

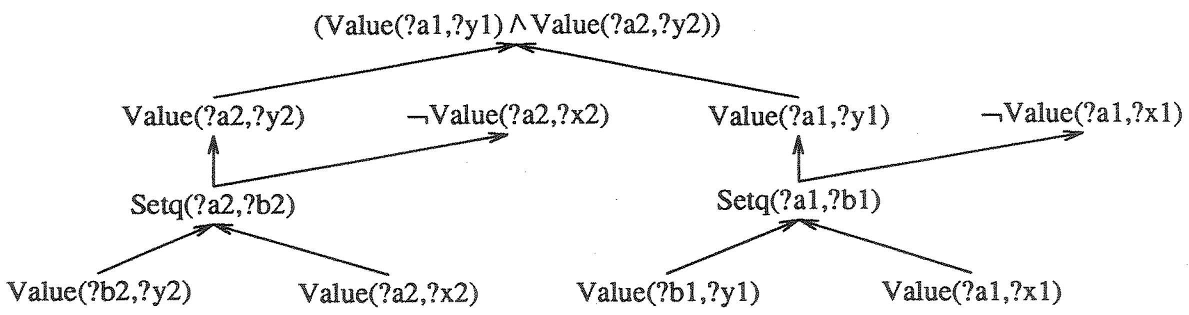

Figure 2: Variable Assignment Example -- Generalized Explanation

presented to the system. Generalization is accomplished by maintaining unifications which connect together general rules composing the explanation while eliminating unifications to facts about the specific situation. For example, for plan-based explanations composed of STRIPS operators, EGGS maintains only unifications connecting effects of one action to the preconditions of other actions. For each representation language, the system also has modules for packaging generalized explanations into macro-rules or macro-operators which can then be used to efficiently solve similar problems in the future. EGGS has been tested on examples from domains such as "blocks world" planning, logic circuit design, artifact classification, and various forms of mathematical problem solving and theorem proving.

4. Generating Partially-Ordered Macro-Operators

The general problem of learning a partially ordered macro-operator as currently solved by the EGGS system can be specified as follows:

Given: Definitions for a set of STRIPS operators and a specific plan composed of instances of these operators that achieves a specific goal.

Determine: A macro-operator for the plan with the most general partial ordering of its operators such that all connections between preconditions and effects present in the original instance are maintained and such that the macro-operator achieves a generalization of the goal achieved by the original instance. Maintaining these two conditions is referred to as maintaining the *explanation structure* of the original plan.

The algorithm implemented in EGGS for determining a partially-ordered macro-operator can be broken down into the following four steps:

(1) Use the EGGS explanation generalization algorithm to produce a generalized explanation for the specific plan.

(2) Determine the minimal set of ordering constraints on the operators in the generalized explanation such that the explanation structure is maintained.

(3) Detect subsets of these ordering constraints which are inconsistent and add preconditions to the macro-operator to prevent such inconsistencies.

(4) Determine the overall preconditions and effects of the final macro-operator.

The first step in this algorithm is standard explanation-based generalization as described by Mitchell *et al.* (1986) and Mooney & Bennett (1986). The generalized explanation EGGS produces for the Setq example is shown in Figure 2. The remaining steps are described in detail in the following subsections.

4.1. Determining a Minimal Set of Ordering Constraints

Most ordering constraints in a plan are imposed by one action achieving a precondition for another action. These ordering constraints are captured by the explanation structure of a plan: an action must precede another action if it eventually supports it. Any topologically sorted list of the actions in the generalized explanation graph will be an ordering that satisfies these constraints.[1] In the Setq example, the generalized explanation shown in Figure 2 does not impose an ordering on the operators and both permutations are topologically sorted. However, as in this example, some topologically sorted orderings may, in certain situations, result in protection violations in which an action deletes or *clobbers* the ultimate goal or a precondition for a later action.

Preventing protection violations involves determining which facts can be deleted in the general case and detecting deletions of protected facts. Figure 3 shows the algorithm implemented in EGGS which determines deletions for the generalized explanation and imposes additional ordering constraints to prevent protection violations. In the algorithm, *Supports?(a, b)* refers to a function that returns true iff *a* eventually supports *b* in the generalized explanation, i.e. if there is a directed path from *a* to *b* in the explanation graph. For each action, the algorithm determines the set of states in the explanation that could be true before the action is executed based only on the partial ordering of actions imposed by the explanation structure. These are called the *deletable* states since they are ones that the action could possibly delete. For each of these states, the condition (α) under which the action could delete the state in general are determined. For example, if an effect of an action is:

[1] A *topological sort* of a directed acyclic graph is an ordering of the nodes in the graph such that a node *u* must occur before a node *v* in the ordering if there is a directed path from *u* to *v*.

```
for each action a in the generalized explanation do
  let deletable = ∅
  for each state s in the generalized explanation do
    if ¬Supports?(a, s) then let deletable = deletable ∪ {s}
  for each effect e of a do
    for each state d in deletable do
      if ¬d unifies with e
        then
          let α be the conditions under which ¬d and e unify
          let c be the action of which d is an effect (possibly NIL)
          if null(c) ∨ Supports?(c, a)
            then
              if d is the goal or supports a proof of the goal
                then add ¬α to the preconditions for the macro-operator
                else d is deleted by e when α is true
            else
              if d is the goal or supports a proof of the goal
                then a must precede c when α is true
                else d is deleted by e when α is true and c precedes a
          PreventPreconditionClobbering(a, d, c, α)

procedure PreventPreconditionClobbering(a, d, c, α)
(* Add conditions to prevent action a from clobbering precondition d when α is true *)
  for each action b such that d is a precondition of b or supports a proof of a precondition of b do
    if Supports?(a, b)
      then
        if null(c) ∨ Supports?(c, a)
          then add ¬α to the preconditions for the macro-operator
          else a must precede c when α is true
      else
        if null(c) ∨ Supports?(c, a)
          then b must precede a when α is true
          else a must precede c or b must precede a when α is true
```

Figure 3: Algorithm for Determining Deletions and Preventing Protection Violations

¬Value(?x, ?y), then the state: Value(?a, ?b) is deleted iff ?x=?a and ?y=?b. If the deletable state is not supported by another action (i.e. it is a precondition of the macro-operator) or if it is an effect of an action that must precede the deleting action because it supports it, then the state must be true before the action is executed and it will be deleted when α is true. If the deletable state is also the goal or supports a proof of the goal, then $\neg\alpha$ is added to the preconditions to prevent the goal from being clobbered. In order to maintain the explanation structure of the original example, the goal must not be deleted. Otherwise the deletable state is marked as being deleted when α is true. If the deletable state is not necessarily true before the action is executed, then it will be deleted when both α is true and the action supporting it precedes the deleting action. If the deletable state is the goal or supports a proof of the goal, then in order to prevent the goal from being clobbered, the deleting action must precede the action supporting the deleted state when α is true.

The procedure PreventPreconditionClobbering adds additional ordering constraints and preconditions to the macro-operator in order to prevent a precondition from being clobbered when action *a* deletes state *d*. Such a precondition deletion would violate the explanation structure. If an effect of action *a* deletes a precondition *p* of action *b*, then in order to avoid a protection violation either *b* must precede *a* or *a* must precede action *c* where *p* is an effect of *c* (Sussman, 1975; Sacerdoti, 1977). However, the existing partial ordering of the actions imposed by the explanation structure may rule out either or both of these possibilities. If both possibilities are ruled out, then $\neg\alpha$ is added to the list of preconditions for the macro-operator to prevent the plan from being used in such situations. Otherwise, the possible ordering constraints that will prevent the protection violation are recorded.

For the double variable assignment example, the ordering constraints imposed by this algorithm are shown below:

1) Setq(?a1,?b1) should precede Setq(?a2,?b2) when (?a2=?b1 $\wedge$?x2=?y1) or else $\neg$Value(?a2,?x2) will clobber Value(?b1,?y1)

2) Setq(?a1,?b1) should precede Setq(?a2,?b2) when (?a2=?a1 $\wedge$?x2=?x1) or else $\neg$Value(?a2,?x2) will clobber Value(?a1,?x1)

3) Setq(?a2,?b2) should precede Setq(?a1,?b1) when (?a2=?a1 $\wedge$?x2=?y1) or else $\neg$Value(?a2,?x2) will clobber goal: Value(?a1,?y1)

4) Setq(?a2,?b2) should precede Setq(?a1,?b1) when (?a1=?b2 $\wedge$?x1=?y2) or else $\neg$Value(?a1,?x1) will clobber Value(?b2,?y2)

5) Setq(?a2,?b2) should precede Setq(?a1,?b1) when (?a1=?a2 $\wedge$?x1=?x2) or else $\neg$Value(?a1,?x1) will clobber Value(?a2,?x2)

6) Setq(?a1,?b1) should precede Setq(?a2,?b2) when (?a1=?a2 $\wedge$?x1=?y2) or else $\neg$Value(?a1,?x1) will clobber goal: Value(?a2,?y2)

4.1.1. Time Complexity of Determining Ordering Constraints

The worst-case time complexity of the algorithm in Figure 3 is relatively easy to determine. In the following discussion, loops will be referred to by their iterative variables (e.g. the *a* loop, the *e* loop, etc.).

Let the number of nodes in the generalized explanation graph be *n*. Since the function *Supports?* determines whether or not there is a directed path from one node to another in this graph, in the worst case it must traverse the entire graph. Since traversing a graph is $O(|V|+|E|)$ where $|V|$ is the number of nodes and $|E|$ is the number of edges and since $|E|\leq|V|^2$, the worst case complexity of a call to *Supports?* is $O(n^2)$.

Since the number of actions must be less than *n*, the outermost loop (the *a* loop) is executed at most *n* times. Since the number of states is also less than *n*, the body of the loop for determining the set of deletable states (the *s* loop) is executed at most *n* times for each action for a maximum total of n^2 times. Since the body of this loop includes a call to *Supports?* which is $O(n^2)$, the total maximum time needed for all executions of the *s* loop is $O(n^4)$. Since each deletable set is a subset of the set of states, its maximum cardinality is *n*.

Since the total number of effects of actions must be less than the number of states, the body of the *e* loop is executed at most a total of *n* times. Since *n* is the maximum size of a deletable set, the body of the *d* loop is executed at most *n* times for each effect for a maximum total of n^2 times. In addition to the call to PreventPreconditionClobbering, the operations in the body of the *d* loop that depend on the size of *n* include a call to *Supports?* and determining whether a state supports a proof of the goal. Since both of these operations involve finding a path between two nodes in the explanation graph, they take at most $O(n^2)$ time. Therefore, let the total maximum time needed for all executions of the *e* loop be:

$$n^2(O(n^2) + P)$$

where *P* is the time complexity of a call to PreventPreconditionClobbering.

The body of the *b* loop in PreventPreconditionClobbering is executed less than *n* times since the *total* number of actions is less than *n*. Since executing the body of the loop requires two calls to *Supports?* which takes $O(n^2)$ time, the maximum time required for a call to this function is $O(n^3)$. Therefore, the total maximum time needed for all executions of the *e* loop is:

$$n^2(O(n^2) + O(n^3)) = O(n^5)$$

Since, the total maximum time needed for all executions of the *s* loop was calculated to be only $O(n^4)$, the worst case time complexity of the complete process is also $O(n^5)$.

The fact that the $O(n^2)$ *Supports?* function is called within bodies of loops that are executed $O(n^2)$ and $O(n^3)$ times is a dominant factor in the complexity of the complete algorithm. If a call to this function could be reduced to constant time, a minor modification of the preceding analysis demonstrates that the complexity of the complete process would then be $O(n^3)$. The maximum time required for all executions of the *s* loop becomes $O(n^2)$. The total maximum time for all executions of the *e* loop becomes:

$$n^2(c + P)$$

where *c* is a constant. The time complexity of a call to PreventPreconditionClobbering (*P*) becomes $O(n)$ and the overall worst case complexity becomes:

$$O(n^2) + n^2(c + O(n)) = O(n^3)$$

Since *Supports?(a,b)* simply determines whether or not there is a directed path from *a* to *b*, the value of this function for all pairs of the *n* nodes in the explanation graph can be pre-computed by determining the graph's *transitive closure*.[2] Since there exist $O(|V|^3)$ algorithms for computing the transitive closure of a directed graph (Reingold, Nievergelt, & Deo, 1977; Sedgewick, 1983) all the values of *Support?* for a particular explanation can be pre-computed in $O(n^3)$ time. Calls to this function will then take constant time and the complete process of determining deletions and protection violations can be accomplished in $O(n^3)$ time.

[2]The *transitive closure* of a directed graph G=(V,E) is a directed graph G′=(V,E′) such that there is an edge $u \rightarrow v$ in E′ if and only if there is a (non-empty) directed path from *u* to *v* in E.

In conclusion, although the algorithm implemented in EGGS is $O(n^5)$, computation of the the transitive closure of the generalized explanation graph using one of the known $O(|V|^3)$ algorithms would result in an $O(n^3)$ algorithm for the problem of determining deletions and preventing protection violations. In any case, computing the minimal set of ordering constraints is a relatively efficient process which can be done in time which is polynomial in the size of the plan.

4.2. Detecting and Preventing Inconsistent Ordering Constraints

The ordering constraints posted to prevent protection violations may contradict each other in certain situations. For example, in the Setq example, when ?a2=?b1, ?x2=?y1, ?a1=?b2, and ?x1=?y2, constraints 1 and 4 shown above contradict each other since they each specify a different ordering of the two operators. With regards to assignment statements, this situation corresponds to swapping the values of two variables (e.g. Setq(A,B), Setq(B,A)) which cannot be done using only two variable assignments. In order to detect such situations, a resolution theorem prover is used to determine sets of ordering constraints that are inconsistent.[3] In addition to the posted ordering constraints, the theorem prover is given the following two axioms:

Before(?a,?b) $\wedge$ Before(?b,?c) $\rightarrow$ Before(?a,?c)
Before(?a,?b) $\rightarrow$ $\neg$Before(?b,?a)

When the resolution theorem prover finds a contradiction, it returns the set of axioms upon which the proof depends. In order to prevent such a contradiction from arising during the subsequent use of the macro-operator, an additional precondition must be added. Assuming each of the n ordering constraints, c_i, in the contradictory set must be satisfied when condition α_i is true, the following precondition must be added to the macro-operator: $\neg(\alpha_1 \wedge \alpha_2 \dots \wedge \alpha_n)$. For the variable swapping example above, the added precondition would be:

$\neg$((?a2=?b1 $\wedge$?x2=?y1) $\wedge$ (?a1=?b2 $\wedge$?x1=?y2))

The system simplifies this precondition to: $\neg$(?a2=?b1 $\wedge$?a1=?b2) using the following domain axiom: Value(?x,?a) $\wedge$ Value(?y,?b) $\wedge$?x=?y $\rightarrow$?a=?b. This axiom simply states that a variable has only one value at a time.[4] A similar contradiction detected between constraints 2 and 5 results in the addition of the precondition: $\neg$?a1=?a2. This situation occurs when both assignment statements are attempting to set the same variable.

Of course, using a resolution theorem prover to find all sets of contradictory axioms is a computationally intractable process which is not even guaranteed to halt. In EGGS, the theorem prover is given a time limit, after which it stops and returns proofs for all the contradictions it has found so far. Consequently, the system does not actually guarantee the prevention of ordering contradictions during later instantiation. Should such a contradiction arise, one could perform explanation-based learning from failure to modify the plan at that time in order to prevent similar problems in the future. Gupta (1987) and Chien (1987) describe implemented systems that use EBL to learn from failures due to unforeseen

[3] Because of the possible presence of disjunctive ordering constraints like: Before(A,C) $\vee$ Before(B,A), a simple cycle-detecting algorithm cannot be used to find all sets of inconsistent constraints. In fact, disjunctive constraints seems to rule out a polynomial time solution to this problem.

[4]Mooney (1988a) gives details on the procedure used to simplify the preconditions of a macro-operator.

interactions.

4.3. Assembling the Partially-Ordered Macro-Operator

Assembling the final macro-operator is a relatively simple task. The preconditions of the macro-operator include the leaves of the explanation as well as any additional constraints added during the prevention of protection violations or inconsistent ordering constraints. If an effect of an individual operator in the plan was never marked as deleted under any condition, then it is added to the list of effects of the macro-operator. If an effect was marked as being deleted under some set of conditions, then the overall effect is given by the following implication: if none of the deletion conditions are met, then the state is true (Fikes, Hart, & Nilsson, 1972). Finally, the posted ordering constraints are added to macro-operator with the exception that constraints whose condition contradicts a precondition of the macro-operator are eliminated. In the example, ordering constraints 2, 3, 5, and 6 can be eliminated since they require that ?a1=?a2, a condition which is now prevented by a precondition.

The final partially-ordered macro-operator that EGGS learns from the "double Setq" example is shown in Table 2. In summary, the system notices that if the two variables being set are the same, then the goal of having them set to two different values cannot be achieved. Also, if either assignment references the variable set by the other, then the two cannot be executed in parallel. In these cases, the assignments must be properly ordered to prevent the referenced variable from being reset before it is used. Finally, if each assignment references the variable set by the other, then two assignments cannot solve the problem. This is the classic ''variable swapping'' problem which requires the use of a temporary variable.

5. Relation to Nonlinear Planning

Work in *nonlinear planning* (Sacerdoti, 1977) has addressed the issue of building plans with partially ordered actions by initially assuming there are no goal interactions and then detecting protection violations and imposing ordering constraints to prevent them. Although the task of building a specific partially-ordered plan for achieving a specific goal is quite different from the task of generalizing a specific totally-ordered plan into a general partially-ordered macro-operator, some of the underlying processes are quite similar.

Nilsson (1980) uses the term DCOMP to refer to the procedure for detecting and preventing protection violations in nonlinear planning. The procedure in Figure 3 is in some ways more general and in some ways less general than DCOMP. It is more general because it must determine the conditions (α) under which a deletion will take place in the generalized

Table 2: Macro-Operator Learned from the Variable Assignment Example

SetqSetq(?a1,?b1,?x2,?y2,?x1,?y1,?a2,?b2)		
Preconditions	Effects	Orderings

Value(?a1,?x1)
Value(?a2,?x2)
Value(?b1,?y1)
Value(?b2,?y2)
?a2≠?a1
¬(?a1=?b2 ∧ ?a2=?b1)
	Value(?a1,?y1)
	Value(?a2,?y2)
	¬Value(?a1,?x1)
	¬Value(?a2,?x2)
		?a1=?b2 → Before(Setq(?a2,?b2),Setq(?a1,?b1))
		?a2=?b1 → Before(Setq(?a1,?b1),Setq(?a2,?b2))

plan. DCOMP works with a specific instance of a plan and therefore does not need to determine which instantiations of a plan result in a deletion. The procedure in Figure 3 is less general than DCOMP because it does not consider alternative ways of achieving the preconditions of an action. The explanation structure determines how preconditions were met in the particular example, and these constraints are retained in the generalization. Using Nilsson's terminology, the procedure presented here does not determine all of the possible *adders* of a precondition in the plan since each precondition is assumed to added by the action that added it in the original specific instance. Of course, an even more general macro-operator could be produced by generalizing the explanation structure and considering alternative adders. However, this would even further complicate the procedure for generating a macro-operator.

The actual process of determining what Nilsson calls a *noninteractive* ordering for a particular instantiation of a plan must be done when the final macro-operator is used to solve a future problem. Assuming all inconsistent sets of ordering constraints were detected and prevented by adding additional preconditions to the macro-operator, for any situation that meets the preconditions, there is guaranteed to be a partial ordering of the actions that satisfies all of the ordering constraints. However, due to the possible presence of disjunctive ordering constraints, actually finding it may require an expensive search. However, this should not be surprising since the increased generality of a partially-ordered macro-operator can clearly result in a corresponding decrease in operationality (Segre, 1987).

6. Another Example

An additional example illustrating the importance of detecting protection violations when generalizing the order of operators in a macro-operator involves learning a schema for "arson for insurance." This is a concept learned by GENESIS, a text understanding system which uses EGGS to learn plan schemata by generalizing specific plans it observes in natural language narratives (Mooney, 1988). By understanding and generalizing the following story, GENESIS acquires a general schema for someone burning their own building in order to collect the insurance money.

> Stan owned a warehouse. He insured it against fire for 100000 dollars. Stan burned the warehouse. He called Prudential and told them it was burnt. Prudential paid him 100000 dollars.

The English paraphrase the system generates for the schema it learns from this story follows:

> ?a3 is a person. ?o2 is an inanimate object. ?c3 is an insurance company. ?v3 is money. ?o2 is not burnt. ?a3 has ?o2. ?a3 insures ?o2 with ?c3 for ?v3 in case it is burnt. ?o2 is flammable. ?a3 burns ?o2. ?a3 contacts ?c3 and tells it that ?o2 is burnt. ?c3 indemnifies ?a3 ?v3 for the loss of ?o2.

The explanation for this plan is basically that both the burning and insuring actions achieve preconditions for the indemnify action which achieves a character's goal of possessing money. However, the structure of this explanation does not impose an ordering on the burning and insuring actions since neither of these actions enables the other in any way. However, the burning action clobbers a precondition of the insuring action since an object cannot be insured against fire if it is already burnt. The algorithm described above detects this potential protection violation and imposes the following mandatory ordering constraint on these two actions:

?a3 insures ?o2 with ?c3 for ?v3 in case it is burnt. should proceed ?a3 burns ?o2. when T or else ?o2 is burnt. will clobber ?o2 is not burnt.

In this example, the final schema is no more general that which would be learned if order generalization were not performed. However, since order generalization was attempted, the system has an explanation for why the insuring action *must* proceed the burning action. The ordering of the actions in the generalized schema is justified by the existing theory of the domain, which it would not be if order generalization where not attempted. Therefore the current approach more completely achieves the goal of producing a justified generalization, which is one of the primary goals of explanation-based methods (Mitchell *et al.*, 1986).

7. An Example Requiring Structural Generalization

As discussed earlier, the order-generalization process currently incorporated in EGGS requires that preconditions be achieved as they were in the original example. However, generalizing the order of operators in certain situations requires altering the explanation structure by breaking some of the connections between preconditions and effects. For example, consider the following plan for building two independent stacks of blocks:

Pickup(A), Stack(A,B), Pickup(C), Stack(C,D)

Assuming the standard definitions for these blocks world actions (Nilsson, 1980), the specific explanation for this plan is shown in Figure 4. Although the process of building two separate stacks would seem to be two independent operations either of which could be performed first, the structure of this explanation actually imposes a particular ordering on the two operations. This is because a precondition for building the second stack is that the hand be empty and this precondition was met by releasing the top block after building the previous stack. Generalizing the order of these two operations requires allowing the HandEmpty preconditions to be achieved in a manner different from that observed in the original instance.

The current order-generalization routine in EGGS does not allow altering the structure of the original explanation by breaking the connection between effects and preconditions. Unlike the ''double Setq'' example discussed earlier, the explanation structure for this example imposes a total ordering on the actions in the plan. Therefore the system cannot generalize the order of these two block-stacking operations. More complex order-generalization techniques which consider the possibility of altering the explanation structure are needed to perform generalizations of this sort.[5]

In the block stacking example, although having the hand empty is a precondition for picking up block C, making the hand empty is not the ''reason'' for stacking A on B. On the other hand, the reason for picking up block A is to enable stacking it on B. Therefore, not all connections between effects and preconditions are created equal. Some seem to be more crucial to the overall explanation than others. One possible rule is to only allow breaking the connection between an effect and a precondition if another effect of the same action more directly supports the goal. This is the situation in the block stacking example since the real reason for stacking A on B is not to enable Pickup(C) but rather to achieve On(A,B).

[5]Mooney (1988b) presents more information on the relationship between order generalization and structural generalization.

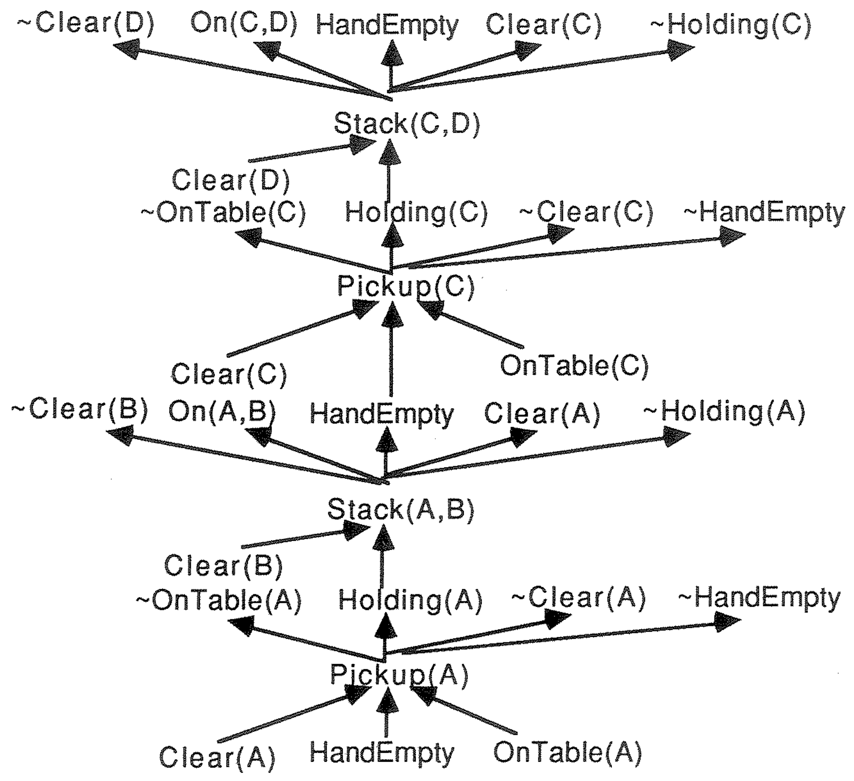

Figure 4: Block Stacking Example -- Specific Explanation

8. Conclusions and Problems for Future Research

Generalizing the temporal order of operators in a plan in order to produce partially-ordered macro-operators can result in more general and as well as more justified concepts. As in the EGGS system, performing this task can be incorporated as part of a general explanation-based learning system. Empirical testing of this system on a number of examples from various domains such as programming, blocks-world planning, and narrative understanding has given promising results.

A theoretical analysis of the computational complexity of the order-generalization problem demonstrated that detecting potential protection violations in macro-operators and determining ordering constraints which prevent them can be performed in polynomial time. However, detecting inconsistencies in these constraints and adding preconditions to prevent them can be computationally expensive. Avoiding a complete search for inconsistencies could result in the omission of necessary preconditions and the resulting acquisition of macro-operators which are overly-general. A promising area for future research involves developing failure-driven EBL techniques for detecting overlooked inconsistencies encountered during subsequent use of such macro-operators and refining the macro-operator to account for the detected interaction.

In addition, as discussed in the previous section, techniques are needed for allowing order generalizations which require altering the structure of the original explanation by breaking the connections between preconditions and effects. Finally, the current technique requires actions to be represented as STRIPS operators with complete delete lists. This is frequently an unrealistic requirement and techniques are needed for generalizing the order of weaker types of operators.

Acknowledgements

This paper presents research conducted at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign and was supported by the Office of Naval Research under grant N-00014-86-K-0309.

References

Chien, S.A. (1987), "Simplifications in Temporal Persistence: An Approach to the Intractable Domain Theory Problem in Explanation-Based Learning," M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, IL. August. (Also appears as Technical Report UILI-ENG-87-2203, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.)

DeJong, G.F. & Mooney, R.J. (1986), "Explanation-Based Learning: An Alternative View," *Machine Learning* 1, 2, pp. 145-176.

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

Gupta, A. (1987), "Explanation-Based Failure Recovery," *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, July, pp. 606-610.

Korf, R.E. (1985), "Macro-Operators: A Weak Method for Learning," *Artificial Intelligence 26*, pp.35-77.

Laird, J., Rosenbloom, P. & Newell, A. (1986), "Chunking in SOAR: An Anatomy of a General Learning Mechanism," *Machine Learning 1*, 1, pp. 11-46.

Minton, S.N. (1985), "Selectively Generalizing Plans for Problem Solving," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August, pp. 596-599.

Mitchell, T.M., Keller, R., & Kedar-Cabelli, S. (1986), "Explanation-Based Generalization: A Unifying View," *Machine Learning* 1, 1, pp. 47-80.

Mooney, R.J. (1988a), "A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, January. (Also appears as Technical Report UILU-ENG-87-2269, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.).

Mooney, R.J. (1988b), "Generalizing the Order of Operators and its Relation to Generalizing Structure," *Proccedings of the AAAI Spring Symposium Series: Explanation Based Learning*, Stanford, CA.

Mooney, R.J. & Bennett, S.W. (1986), "A Domain Independent Explanation-Based

Generalizer," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August, pp. 551-555.

Mooney, R.J. & DeJong, G.F. (1985), "Learning Schemata for Natural Language Processing," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August, pp. 573-580.

Nilsson, N.J. (1980), *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, CA.

Reingold, E.M., Nievergelt, J., Deo, N. (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ.

Sacerdoti, E. (1977), *A Structure for Plans and Behavior*, American Elsevier, New York.

Sedgewick, R. (1983), *Algorithms*, Addison-Wesley, Reading, MA.

Segre, A.M. (1987), "On the Operationality/Generality Trade-off in Explanation-Based Learning," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August.

Shavlik, J.W. & DeJong, G.F. (1987), "BAGGER: An EBL System that Extends and Generalizes Explanations," *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, July, pp. 516-520.

Sussman, G.J. (1975), *A Computational Model of Skill Acquisition*, American Elsevier, New York, NY.