Ph.D. Proposal, Department of Computer Sciences, University of Texas at Austin, September 1996
Also appears as Technical Report AI 96-250, Artificial Intelligence Lab, University of Texas at Austin

Integrating Explanation-Based and Inductive Learning Techniques to Acquire Search-Control for Planning*

Tara A. Estlin
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
estlin@cs.utexas.edu

Technical Report AI96-250
September 1996

Abstract

Planning systems have become an important tool for automating a wide variety of tasks. Control knowledge guides a planner to find solutions quickly and is crucial for efficient planning in most domains. Machine learning techniques enable a planning system to *automatically* acquire domain-specific search-control knowledge for different applications. Past approaches to learning control information have usually employed *explanation-based learning* (EBL) to generate control rules. Unfortunately, EBL alone often produces overly complex rules that actually decrease rather than improve overall planning efficiency. This paper presents a novel learning approach for control knowledge acquisition that integrates explanation-based learning with techniques from *inductive logic programming*. In our learning system SCOPE, EBL is used to constrain an inductive search for control heuristics that help a planner choose between competing plan refinements. SCOPE is one of the few systems to address learning control information for newer, partial-order planners. Specifically, this proposal describes how SCOPE learns domain-specific control rules for the UCPOP planning algorithm. The resulting system is shown to produce significant speedup in two different planning domains, and to be more effective than a pure EBL approach. Future research will be performed in three main areas. First, SCOPE's learning algorithm will be extended to include additional techniques such as constructive induction and rule utility analysis. Second, SCOPE will be more thoroughly tested; several real-world planning domains have been identified as possible testbeds, and more in-depth comparisons will be drawn between SCOPE and other competing approaches. Third, SCOPE will be implemented in a different planning system in order to test its portability to other planning algorithms. This work should demonstrate that machine-learning techniques can be a powerful tool in the quest for tractable real-world planning.

*This research was supported by the NASA Graduate Student Researchers Program, grant number NGT-51332.

1 Introduction

In the past few decades, planning systems have become a powerful tool for automatically performing many daily tasks. Given a set of goals, these systems are designed to output a list of actions that can be used by an execution agent to perform a task with little or no human intervention. Planning systems are currently used to perform a range of activities from manufacturing semiconductors (Fargher & Smith, 1994) to scheduling antenna communications with orbiting spacecraft (Chien et al., 1995). As the desire for automation grows more prevalent in today's society, the need for efficient planning systems grows as well.

Artificial intelligence researchers have introduced numerous approaches to planning, however, even the newest domain-independent planning algorithms require large amounts of computation and cannot tractably handle most realistic problems. Researchers have recognized that efficient planning requires incorporating considerable amounts of specific knowledge about a domain. This additional domain information is often termed "control knowledge" and indicates *how* a planner should best achieve its goals. Control knowledge helps a planner to avoid considerable search by directing it towards the most promising search paths early in the planning process. This knowledge is usually represented in the form of search-control rules that can be easily utilized by the planner. Without the addition of control rules, a planner may be unable to produce a solution in reasonable time.

Unfortunately, constructing control heuristics for a new domain is a difficult, laborious task. Research in planning and learning attempts to address this problem by developing methods that automatically acquire search-control knowledge for different domains. By examining past planning scenarios, a learning system can acquire control information that will help the planner perform well on future problems. Most past systems have employed *explanation-based learning* (EBL) to acquire search-control information. Unfortunately, standard EBL often produces complex, overly-specific control rules that decrease rather than improve overall planning performance (Minton, 1988).

This proposal presents a novel learning algorithm for automatically acquiring search-control knowledge for planning systems. Our learning system SCOPE uses a unique combination of machine learning techniques to acquire effective search-control rules for planning. By incorporating induction with EBL to learn simple, approximate control rules, we can greatly improve the utility of acquired knowledge (Cohen, 1990; Leckie & Zuckerman, 1993). Specifically, SCOPE (Search Control Optimization of Planning through Experience) integrates *explanation-based generalization* (EBG) (Mitchell et al., 1986; DeJong & Mooney, 1986) with techniques from *inductive logic programming* (ILP) (Quinlan, 1990; Muggleton, 1992; Lavrač & Džeroski, 1994) to learn high-utility rules that can generalize well to new planning situations.

SCOPE extends previous planning and learning research by acquiring control knowledge for the newer, more efficient *partial-order* planners. Most work in learning control rules for planning has been in the context of linear, state-based planners (Minton, 1989; Gratch & DeJong, 1992; Leckie & Zuckerman, 1993). These planners are usually classified as *total-order* planners, since plans steps are maintained in a strictly ordered list. Recent experimental results, however, support that partial-order planners are more efficient than total-order

planners in most domains (Barrett & Weld, 1994; Kambhampati & Chen, 1993; Minton et al., 1992). In partially-ordered plans, some steps can remain unordered with respect to each other, thereby allowing a planner to avoid premature commitments to an incorrect ordering. Though partial-order planners are considered a more proficient planning strategy, they are not a panacea for efficient planning; added control knowledge can still dramatically effect their performance. Unfortunately, there has been little work on learning control rules for partial-order planning systems (Katukam & Kambhampati, 1994).

SCOPE learns control rules in the form of selection heuristics. These heuristics greatly reduce backtracking by directing a planner to immediately select appropriate plan refinements. SCOPE is implemented in Prolog, which provides a good framework for learning control knowledge. Many different problem solvers can be easily coded as Prolog programs and control knowledge is easy to incorporate. A version of the UCPOP planning algorithm (Penberthy & Weld, 1992) was implemented as a Prolog program to provide a testbed for SCOPE. In this paper, experimental results are presented on two planning domains that demonstrate SCOPE can significantly increase partial-order planning efficiency. SCOPE is also shown to outperform a competing approach based only on EBL.

The remainder of this proposal is organized as follows. A background on planning, learning, and inductive logic programming is presented in Section 2. Section 3 describes the UCPOP planner and how control rules are implemented. In section 4, SCOPE's learning algorithm is explained. Section 5 presents and analyzes empirical results of applying SCOPE to two different planning domains. Section 6 discusses related work in learning control information. Section 7 outlines our future research plan, which includes applying SCOPE to new real-world domains and extending SCOPE for use on other planning algorithms. The final section summarizes the goals of this research and its significance.

2 Background

2.1 Planning Basics

A planning problem is traditionally defined as consisting of three main parts: an *initial-state*, a set of *goals*, and a set of possible *actions*. The solution to a planning problem is an ordered list of these actions that will transform the initial world state into a state where the goals are satisfied. The set of available actions depends on the particular task or problem domain. Most planners operate by beginning with a null plan and then adding actions one at a time until a solution is found. A planner must be able to perform a number of functions, including choosing a good action to add to the current plan, detecting when a solution has been found, and recognizing dead-end search paths that should be abandoned.

In most standard approaches to planning, domain actions (or operators) are represented in a STRIPS format (Fikes & Nilsson, 1971), which consists of a list of preconditions, an add list and a delete list. For example, the four operator schemas from the well-known blocksworld domain (Nilsson, 1980) are shown in Figure 1. In order for an action to be executed, its preconditions must be satisfied in the current state of the world. Once an action is applied, any conditions on its add list are added to the current world state and all delete

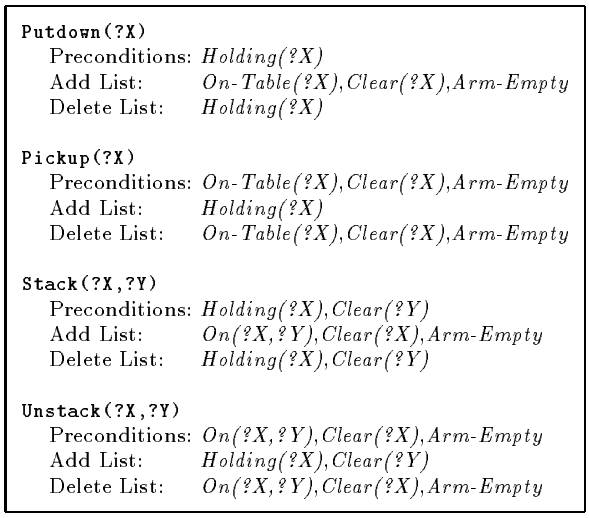

Figure 1: Operator Schemas from the Blocksworld

conditions are removed. In other words, an action's add and delete list describe how the action changes the world. Unfortunately, this action format is insufficiently expressive for most realistic planning domains. In order to perform planning in real-world scenarios, the STRIPS-style action format has been extended by many researchers to include more expressive constructs. Additional constructs include items such as conditional effects, universal quantification of precondition and effect variables, and disjunctive preconditions (Pendault, 1989; McDermott, 1991; Penberthy & Weld, 1992; Chien & DeJong, 1994).

Given a planning problem, many different search methods have been developed for finding a correct plan. Most classical planners, including the original STRIPS planner, employ a *state-based* search. In this approach, the planner begins with the problem initial state and attempts to apply operators until the desired goal state is reached. To help guide its search, the planner always maintains a representation of the current world state. The planning process can involve either a simple forward search from the initial state or a more sophisticated goal-directed search that reasons backwards from the goal state. A well-known goal-directed search technique is *means-end analysis*, which selects operators that reduce the difference between the goal state and the current state. This type of planning algorithm is often termed a *total-order* approach since the current plan must be maintained as an *ordered* list of actions.

A second style of planning employs a *plan-based* search and does not maintain the current world state as it proceeds. In this type of planner, operators are still added to the current plan until a solution is found. However, instead of saving the plan as an ordered list of operators, the current plan is represented as a *partially-ordered* set of operators. Only necessary ordering decisions are saved in the plan; all others are postponed until later in the planning process when more information will be available. This type of planner typically proceeds by repeatedly identifying an unachieved goal or precondition and then selecting an operator that will achieve it. Though only necessary ordering constraints are included in the current plan, a valid ordering of all operators must always exist. The planning cycle continues until all goals have been achieved. This type of planner is often referred to as a *partial-order*

planner as opposed to the *total-order* planners discussed above. Many researchers consider partial-order planning a more powerful and efficient planning strategy since premature ordering commitments are delayed until a more informative ordering decision can be made. (Barrett & Weld, 1994; Kambhampati & Chen, 1993; Minton et al., 1992)

One other important distinction that is often made between planning algorithms is whether a planner is *linear* or *nonlinear*. Over the years, these terms have acquired several different connotations and are often confused or mistaken with *state-based* or *plan-based*. Linear planners, which are usually equated with state-based planners, typically examine goals in a "linear" order; if two (or more) goals exist, then the first goal and all of its subgoals must be achieved before the next goal is considered. In contrast, nonlinear planners do not employ the linearity restriction and can examine goals in any order. Partial-order planners are typically nonlinear planners, however they are not required to be. There are also nonlinear planning algorithms that employ a state-based approach (Warren, 1974; Veloso et al., 1995). Since the terms *linear planner* and *nonlinear planner* are not well-defined, many recent authors choose instead to refer to planners as either *total-order* or *partial-order*.

Unfortunately, even with the most current algorithms, most planning problems are difficult to solve. Even toy domains, such as the well-known blocksworld, quickly become computationally intractable as problem difficulty increases. This intractability is further aggravated by more expressive domain definitions. Though constructs such as universal quantification and conditional effects are important domain representation tools, they greatly contribute to planning complexity. In order to make planning on many domains more tractable, researchers often attempt to utilize specific knowledge about a domain (Fikes & Nilsson, 1971; Minton, 1989; Langley & Allen, 1991; Kambhampati et al., 1996). This knowledge can provide a planner with the extra information needed to make wise decisions early in the planning process, thereby avoiding large amounts of search.

This additional domain knowledge is usually termed "control" knowledge and is specified in the form of search-control rules or heuristics. Control rules are used to direct how a planner searches for a plan. Rules can make decisions such as what goal should be examined next or what action should be used to achieve a particular goal. Unfortunately, acquiring control knowledge for new domains is a difficult task. Eliciting knowledge from a domain expert and coding that knowledge in a usable form can be extremely time-consuming. Plus, for many domains, a domain expert may not be readily available. Control information must also be re-acquired each time the domain changes or a new domain is encountered. One attractive alternative is to allow the system to automatically learn needed control knowledge. This solution is considered in more detail in the following section.

2.2 Learning Methods for Planning

Planning systems can often *automatically* acquire control knowledge by utilizing machine learning techniques. This knowledge can be based directly on user input or derived from past planning experience. For instance, a system which acts as a *learning apprentice* is designed to acquire control information by observing the behavior of a human expert. Other learning systems are built to learn from their own experience without human intervention.

These autonomous systems are understandably more complex and difficult to build, however, they are frequently more desirable since they require little or no human overhead.

Several learning techniques have been applied to control knowledge acquisition. Most common are *analytical* methods where a *domain theory* is used to bias the learner toward useful control information. The most popular of these methods is *explanation-based* learning (EBL) (Mitchell et al., 1986; DeJong & Mooney, 1986). In EBL, examples guide the formulation of a concept but no inductive leap is made. An input domain theory designates a set of "operational" concepts from which learned concepts are drawn. Operational concepts are usually low-level predicates that have been classified as "easy to evaluate" in the problem domain, and thus can be used to build efficient concept definitions. Using the domain theory, a proof is constructed for each example of why that example is a valid member of the target concept. This proof is then generalized and the set of operational concepts contained in the proof are used to build a concept definition. Explanation-based learning and other analytical techniques are considered *knowledge-rich* due to their ability to explain why a particular rule was learned. A number of EBL learning systems have been applied to acquire planning control knowledge (Minton, 1989; Bhatnagar & Mostow, 1994; Kambhampati et al., 1996). Unfortunately, due to its reliance on a only a few examples, standard EBL can often produce complex, overly-specific control rules that do not generalize well to new planning situations (Minton, 1988). Even though the learned rules are correct, the cost of testing their applicability on new planning situations often outweighs their savings. This situation is commonly known as the *utility problem*. EBL methods are also hard to apply in domains where it is difficult to construct a complete and tractable domain theory (Chien, 1989).

At the other extreme are inductive methods which learn through examining examples of positive and negative planning situations. These empirical methods build control rules by making inductive leaps based on some domain-independent bias such as an information-gain heuristic or Occam's razor (Quinlan, 1983). This type of approach obviates the need for a domain theory since rules are learned directly from empirical information. Inductive methods also tend to build more general control rules, and thus often acquire more useful control knowledge. A disadvantage to inductive algorithms is that they usually require large numbers of examples to acquire effective control information. These methods can also easily become computationally intractable since they often search through large amounts of background information when building control rules. This background information usually corresponds to extensional definitions of any concepts that can be used to build control rules. Inductive methods have not received as much attention as EBL, but a few systems have utilized induction for learning search-control knowledge (Mitchell et al., 1983; Porter & Kibler, 1986; Langley & Allen, 1991; Leckie & Zuckerman, 1993).

An alternative approach is to use a combination of learning techniques to acquire control information. Most of these methods attempt to combine EBL with an inductive algorithm where the main goal is to retain the benefits of a domain theory while also having the flexibility to learn from the data. For example, instead of building a complete proof, *plausible explanation-based learning* (PEBL) (Zweben et al., 1992) first conjectures an example is a member of target concept, and then confirms the conjecture with empirical data. Other systems have employed *lazy explanation-based learning* (LEBL) which generates incomplete explanations and then incrementally refines any overly-general knowledge using new planning

examples (Tadepalli, 1989; Borrajo & Veloso, 1994b).

This proposal presents a novel multi-strategy learning approach to control-knowledge acquisition. Our learning system SCOPE also uses a combination of EBL and induction to learn control information. However, instead of generating control rules through EBL and then inductively refining them, SCOPE directly builds rules using an inductive algorithm. EBL is used to focus the inductive search so that only highly relevant pieces of background information are examined. This technique biases the search towards more useful rules, and also keeps the inductive search at a computationally tractable level. SCOPE also differs from other learning approaches by employing techniques from *inductive logic programming* (Muggleton, 1992; Lavrač & Džeroski, 1994). A detailed description of the SCOPE learning algorithm is presented in Section 4.

2.3 ILP Methods for Control Learning

Inductive logic programming (ILP) techniques have been primarily used in the past for inducing logic programs based on a set of examples. Due to the expressiveness of first-order logic, ILP methods can learn relational and recursive concepts that cannot be represented in the attribute/value representations used by most machine-learning approaches. ILP systems have successfully induced small programs for simple tasks such as sorting and list manipulation (Muggleton & Buntine, 1988; Quinlan & Cameron-Jones, 1993); as well as performing well on more complicated tasks such as learning properties of organic molecules (Muggleton et al., 1992) and predicting the past tense of English verbs (Mooney & Califf, 1995).

More recently, it has been argued that ILP techniques can also be a useful tool for acquiring control information (Cohen, 1990). Many different problem solving strategies can be easily coded as Prolog programs and learning mechanisms are also easily implemented in this framework. Logic programs have long been recognized as a good platform for EBL techniques since the notion of "explanation" can be equated with the structure of a proof. On the empirical side, there are a number of current learning systems that employ ILP techniques to induce Horn clause concept definitions (Quinlan, 1990; Muggleton, 1992). By casting the problem of learning control rules as a concept learning problem, these inductive techniques can often be successfully used to acquire control information.

Logic programming also provides a well-understood representation and computation platform upon which to build. A logic program is expressed using the definite clause subset of first-order logic, where a definite clause is a disjunction of literals having exactly one unnegated literal. The one unnegated literal represents the clause *head* while the other literals comprise the clause *body*. Computation in this representation is done using a resolution proof strategy on an existentially quantified goal. For example, a simple logic program to sort lists (written in Prolog) is shown in Figure 2. The top-level goal of this program is `sort(X,Y)`. An instantiation of this goal is true when `Y` is a sorted version of the list represented by `X`. The arguments of a top-level goal are usually partitioned into input and output argument sets. In this example, `X` is considered the input and `Y` the output. A program is executed by providing a goal which has its input arguments instantiated. When a top-level goal is provided, a theorem-prover constructively proves the existence of the goal meeting any constraints

```
sort(X,Y) :-  permutation(X,Y), ordered(Y).

permutation([],[]) :- true.
permutation([X|Xs],Ys) :-  permutation(Xs,Ys1), insert(X,Ys,Ys1).

insert(X,[X|Xs],Xs) :- true.
insert(X,[Y|Ys],[Y|Ys1]) :-  insert(X,Ys,Ys1).

ordered([X]) :- true.
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Figure 2: Simple Sorting Program

provided through the input arguments. In this process, the prover will produce bindings for the output arguments. For example, in our simple sorting program, a top-level goal of the form `sort([6,3,1,5,9],Y)` would produce the output binding `Y = [1,3,5,6,9]`.

The Prolog programming language provides a practical instantiation of logic programming using a simple control strategy. In Prolog, depth-first search with backtracking is used to search for a proof. If during execution the current search path fails, then the last deterministic decision point is backtracked upon and a new path explored. Search control in a Prolog program can be viewed as a *clause selection* problem (Cohen, 1990), where clause selection is the process of deciding what program clause should be used to reduce a particular subgoal during program execution. Different options in a program are represented using separate clauses which have equivalent heads but different clause bodies. Control information is usually incorporated into a Prolog program in the form of clause-selection rules. These rules help avoid inappropriate clause applications, thereby greatly reducing backtracking.

As an example, consider the simple sorting program shown in Figure 2, which sorts a list by generating permutations of the list until it finds one that is ordered. Permutations are generated by permuting the tail of the input list and then inserting the head somewhere in the permuted tail. This program currently performs in $O(N!)$ time. The only nondeterminism comes from the definition of the predicate `insert/3` which can either insert an item at the beginning of a list or somewhere in the tail. This nondeterminism can be eliminated by learning a control rule for the first clause that will correctly predict when the item should be placed at the head of the list.

Figure 3 shows a modified version of the `insert` clause definition, which was constructed by the DOLPHIN learning system (Zelle & Mooney, 1993). The first `insert` clause has been "guarded" with control information so that attempts to use it inappropriately will fail immediately. This clause will now only be applied when a element is being inserted into an empty list or if the new element is less than the head of the current list. The result is an $O(N^2)$ insertion sort program.

Cohen (1990) and Zelle and Mooney (1993) have both introduced systems that acquire control heuristics to improve the performance of Prolog programs. In these systems, a combination of EBL with induction is used to learn control rules that eliminate backtracking. Ex-

```
insert(X,[X|Xs],Xs) :-  insert_control1(X,[X|Xs],Xs).
insert(X,[Y|Ys],[Y|Ys1]) :-  insert(X,Ys,Ys1).

insert_control1(X,[],[X]).
insert_control1(X,[X|Z],[X,Y|Z]) :- X < Y.
```

Figure 3: Improved Insert Predicate

perimental results are presented on a variety of domains including several planning domains which employed a simple linear planner. This proposal presents research that successfully extends these methods by applying ILP techniques for control knowledge acquisition in a modern, complex planning system.

3 Learning Control For Partial-Order Planning

Previous research in planning and learning systems has been based almost entirely around linear, state-based planning algorithms. In recent years, a number of more sophisticated planning approaches have been introduced that outperform most linear, state-based algorithms. Unfortunately, few control-knowledge acquisition systems have been adapted to perform on these newer planning algorithms. One style of planning that has acquired much recent prominence is partial-order planning. This type of approach is widely used in many current planning systems and we felt a partial-order planner would provide an good testbed for our control learning system.

3.1 The UCPOP Planner

The base planner we chose for experimentation is UCPOP (Penberthy & Weld, 1992), a partial-order planner whose step descriptions can include conditional effects and universal quantification. UCPOP has been proven sound and complete, and a significant amount of planning research has been based around its algorithm. Given a planning problem that contains an initial state, a set of goals, and a set of domain operators, the goal of UCPOP is to determine a sequence of operators that will transform the initial state into a state where all goals are satisfied. Operators are specified using a representation similar to the well-known STRIPS format (Fikes & Nilsson, 1971), where operators contain precondition, add and delete lists.[1]

UCPOP searches in a space of partial plans, where each plan consists of a partial-ordering of actions. A partial plan is best described as a four-tuple $\langle \mathcal{S},\mathcal{B},\mathcal{O},\mathcal{L} \rangle$: where $\mathcal{S}$ is a set of actions, $\mathcal{O}$ is a set of ordering constraints over $\mathcal{S}$, $\mathcal{L}$ is a set of causal links, and $\mathcal{B}$ is a set

[1] More specifically, operators are represented in Pednault's Action Description Language (ADL) (Pendault, 1989), which is actually more expressive then STRIPS since it allows several additional constructs such as conditional effects and universal quantification. However, since no domains used in this proposal required these additional constructs we will not consider this point in detail at this time.

of binding constraints over variables appearing in $\mathcal{S}$. Specifically, $\mathcal{S}$ contains all actions that have been added as current plan steps. The set of orderings, $\mathcal{O}$, specifies a partial ordering of these actions. Ordering constraints between steps are usually denoted by the "$<$" relation. For example $S_1 < S_2$ means that step S_1 is constrained to come before S_2. During planning, there must always exist at least one consistent total ordering of all steps.

Causal links, contained in $\mathcal{L}$, record dependencies between the effects of one action and the preconditions of another. A link is represented as $S_1 \stackrel{Q}{\rightarrow} S_2$ where S_1 and S_2 are plan steps and Q is an effect of S_1 and a precondition of S_2. In this case, S_1 is considered the link's producer and S_2 its consumer. These links are used to detect *threats*, which occur when a new action interferes with a past decision. More specifically, if $S_1 \stackrel{Q}{\rightarrow} S_2$ is a causal link in the current plan and there exists a separate action in the plan S_3 which threatens the link, then the following two conditions are satisfied:

- $\mathcal{O} \cup S_1 < S_3 < S_2$ is consistent, and
- S_3 has $\neg Q$ as an effect (or has Q as a delete condition).

When a plan contains a threat, it is possible that it will not work as anticipated. To prevent this from happening, the planner must check for and resolve any discovered threats. UCPOP employs two main threat resolution strategies: *promotion* and *demotion*. For *promotion*, the planner adds an additional ordering constraint to ensure that S_3 is executed before S_1, e.g. $S_3 < S_1$. Similarly, for *demotion* an ordering constraint is added that requires S_3 to be executed after S_2.

Binding constraints are in the form codesignation and noncodesignation constraints. Codesignation constraints represent the required unification of two variables ($?X = ?Y$) or a variable and a constant ($?X = A$). Noncondesignation constraints, conversely, prohibit the unficiation of two variables ($?X \neq ?Y$). These constraints, contained in the set $\mathcal{B}$, apply to variables appearing in the pre- and post-conditions of the actions contained in $\mathcal{S}$.

An overview of the UCPOP algorithm is shown in Figure 4. The algorithm takes three inputs: a plan $\langle \mathcal{S}, \mathcal{B}, \mathcal{O}, \mathcal{L} \rangle$, an agenda of outstanding goals, and a set of action schemata Λ. The initial null plan for a planning problem has two actions, S = $\{A_0, A_\infty\}$, one ordering constraint, $\mathcal{O} = \{A_0 < A_\infty\}$, no causal links, $\mathcal{L} = \{\}$, and no binding constraints, $\mathcal{B} = \{\}$. The initial and goal states are represented in the initial plan by adding the two actions A_0 and A_∞, where the effects of A_0 correspond to the initial state and the preconditions of A_∞ correspond to the desired goal state. The initial agenda contains all top-level goals, where each goal is represented as a pair $\langle Q, A_i \rangle$ where Q is a precondition of A_i.

In each planning cycle, a goal is selected from the agenda in Step 2, and in Step 3, an existing or new action is chosen to assert the goal. After an action is selected, the corresponding orderings, casual links and codesignation constraints are added to $\mathcal{O}$, $\mathcal{L}$, and $\mathcal{B}$, and if a *new* action was selected, it is added to $\mathcal{S}$. Step 4 removes the selected goal from the agenda, and if a *new* action was selected to assert it, that action's preconditions are added to the agenda. Step 5 checks for possible threats and resolves any found by adding an additional ordering constraint through either demotion or promotion. UCPOP is called

Algorithm UCPOP($\langle \mathcal{S}, \mathcal{B}, \mathcal{O}, \mathcal{L} \rangle$, agenda, Λ)

1. **Termination:** If agenda is empty, return $\langle \mathcal{S}, \mathcal{B}, \mathcal{O}, \mathcal{L} \rangle$.
2. **Goal Selection:** Select a goal $\langle Q, A_{need} \rangle$ from the agenda where Q is a precondition of action A_{need}.
3. **Operator Selection:** Choose either an existing action (from $\mathcal{S}$) or a new action A_{add} (instantiated from Λ) that adds Q. Let $\mathcal{O}' = \mathcal{O} \cup \{A_{add} < A_{need}\}$, $\mathcal{L}' = \mathcal{L} \cup \{A_{add} \xrightarrow{Q} A_{need}\}$, and let $\mathcal{B}'$ be the updated set of bindings. If A_{add} is a new action let $\mathcal{S}' = \mathcal{S} \cup A_{add}$ and $\mathcal{O}' = \mathcal{O}' \cup A_o < A_{add} < A_\infty$.
4. **Update Goal Set:** Let agenda' = agenda - $\{\langle Q, A_{need} \rangle\}$. If A_{add} is newly instantiated, then for each condition, Q_i, on its precondition list add $\langle Q_i, A_{add} \rangle$ to agenda'.
5. **Causal Link Protection:** For every action A_t in $\mathcal{S}$ that might threaten a causal link $A_p \xrightarrow{R} A_c$ in $\mathcal{L}$ select a consistent ordering constraint, either
   (a) **Demotion**: Add $A_t < A_p$ to $\mathcal{O}'$, or
   (b) **Promotion**: Add $A_c < A_t$ to $\mathcal{O}'$.

   If neither constraint is consistent then return failure.
6. **Recursive Invocation:** UCPOP($\langle \mathcal{S}', \mathcal{B}', \mathcal{O}', \mathcal{L}' \rangle$, agenda', Λ)

Figure 4: The UCPOP Partial-Order Planning Algorithm[2]

recursively until the agenda is empty. On termination, UCPOP uses the constraints found in $\mathcal{O}$ to determine a total ordering of the actions in $\mathcal{S}$, and returns this as the final solution.

A version of the UCPOP partial-order planning algorithm has been implemented as a Prolog program to provide a testbed for our learning system. Planning decision points are represented in this program as clause-selection problems (i.e. each decision option is formulated as a separate clause). As explained in Section 2.3, this type of representation allows control rules to be easily incorporated into program clauses. Though our planning algorithm is directly based on UCPOP, there are implementation differences. The most significant difference is that our Prolog planner operates using a depth-first backtracking search with a depth bound, while UCPOP normally employs a best-first search strategy.[3] To evaluate the efficiency of our planner as compared to the standard LISP implementation of UCPOP (v2.0) (Barrett et al., 1993), we ran several experiments using problem sets from two domains. These problem sets are also used for testing the learning algorithm, and are discussed in Section 5. In these tests, our planner performed comparably to UCPOP and in some cases, performed better. Though these experiments are not intended to promote a particular search strategy or programming language, they do indicate that our Prolog planner is compatible in terms of efficiency to standard UCPOP.

[2]Algorithm steps involving conditional effects or universal quanitifcation are not shown here. For more details on the use of these constructs see (Penberthy & Weld, 1992).

[3]Kambhampati et al. (1996) also run UCPOP in a depth-first search mode in their control-rule learning system. This system is discussed further in related work.

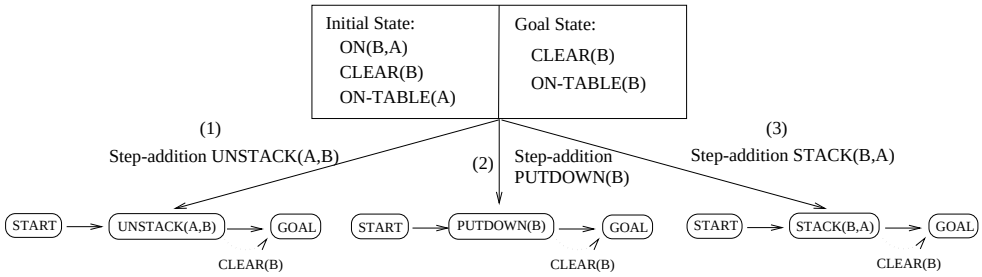

Figure 5: Three competing refinement candidates for achieving the goal *Clear(B)*.

3.2 Planning Decision Points

There are several important decision points in the UCPOP algorithm where the planner must select from one of several possible plan refinements. These decision points include goal selection, goal establishment (selecting an existing or new operator), threat selection, and threat resolution. At these points, there are often a number of valid choices that the planner must choose from. For instance, there may be several actions that can be added to achieve a particular goal. In the absence of control information, our Prolog UCPOP will always select the first valid refinement; other refinements may be tried later through backtracking. An example of a planning decision point from the blocksworld is shown in Figure 5. Here, there are three possible refinement candidates for adding a new action to achieve the goal *Clear(B)*. In this case, only one refinement option (candidate 2) will ever lead to a problem solution.

SCOPE has been designed to only learn control rules for decisions that might lead to a failing search path (i.e. might be backtracked over). Though goal selection and threat selection can affect planner performance, these decisions will *never* be backtracked upon, thus, no control rules are learned for these points. Search control rules for the remaining decisions types (goal establishment and threat resolution) are in the form of refinement-selection rules.

3.3 Control Rule Format

SCOPE learns control rules in the form of selection rules that define when a particular plan refinement should be applied. A selection rule consists of a conjunction of conditions that must all evaluate to true for the refinement to be used. If at least one condition fails, that refinement will be rejected, and the next refinement candidate evaluated. For example, shown below is a selection rule for the first candidate from Figure 5, which contains several control conditions.

Select new-operator `Unstack(?X,?Y)` **to establish** `goal`(*Clear(?Y),*s_1)
  If `exists-operator`(s_2) $\wedge$ `establishes`(s_2,*On(?X,?Y)*) $\wedge$
    `possibly-before`(s_2,s_1).

This rule states that the operator `Unstack(?X,?Y)` should be selected to add *Clear(?Y)* only when there is an existing action s_2 that adds *On(?X,?Y)* and s_2 can be ordered before

the action which requires *Clear(?Y)*, s_1. On the planning decision shown in Figure 5, this rule would fail and thus correctly prevent the planner from selecting the first refinement candidate. Learned control information is incorporated into the planner so that attempts to select an inappropriate refinement will immediately fail. SCOPE can also make selection rules deterministic or nondeterministic depending on the accuracy of the learned rule.

4 The SCOPE Learning System

SCOPE is based on the DOLPHIN speedup learning system (Zelle & Mooney, 1993), which optimizes logic programs by learning clause-selection rules. DOLPHIN has been shown successful at improving program performance in several different domains, including planning domains which employed a simple state-based planner. DOLPHIN, however, has little success improving the performance of a partial-order planner due to the higher complexity of the planning search space. In particular, DOLPHIN's simple control rule format lacks the expressibility necessary to describe complicated planning situations. DOLPHIN also has difficulty successfully generalizing explanations produced by a partial-order planner. Unlike the simple planner DOLPHIN had previously been applied to, a partial-order planner employs many nested data structures to represent plan information. DOLPHIN's learning algorithm could not properly generalize these complex structures so that proof tree information would be usable as control knowledge. SCOPE has greatly expanded upon the original DOLPHIN algorithm to be effective on more complex planning systems.

The input to SCOPE is a planning program and a set of training examples. SCOPE uses the examples to induce a set of control heuristics which are incorporated into the original planner. Figure 6 shows the three main phases of SCOPE's algorithm. First, the example analysis phase solves the training examples using the original planner and extracts useful control information. This information is then used in the control rule induction phase to generate refinement-selection rules. Learned rules are incorporated into the original planner in the program specialization phase. The complete algorithm is explained in detail in the next three sections.

4.1 Example Analysis

In the example analysis phase, two main outputs are produced: a set of selection-decision examples and a set of *generalized* proof trees. Selection-decision examples are used to record successful and unsuccessful applications of a plan refinement. Generalized proof trees provide a background context that explains the success of all correct planning decisions. These two pieces of information are explained in detail below and are used in the next phase to build control rules.

Selection-decision examples are produced using the following procedure. First, training examples are solved using the existing planner. A trace of the planning decision process used to solve each example is stored in a proof tree, where the tree root represents the top-level planning goal and the tree nodes correspond to different planning procedure calls. The top part of a solution tree is shown in Figure 7. The top-level goal in this proof is a call to the

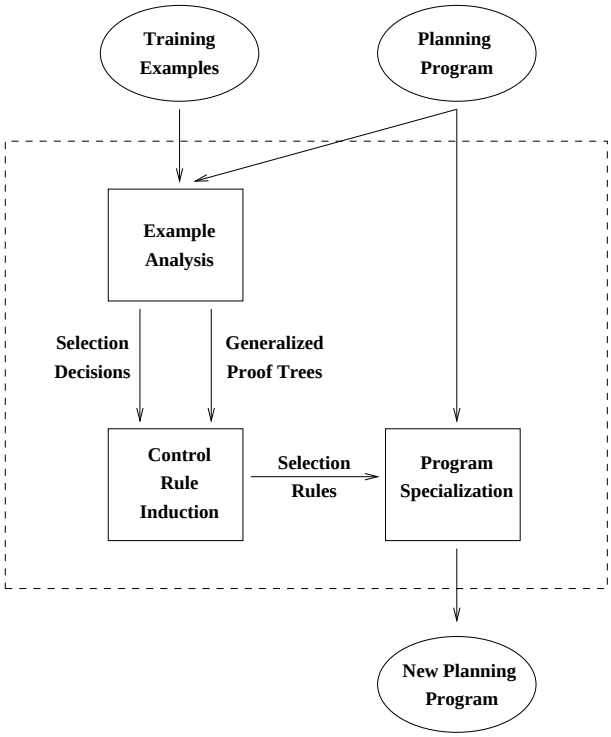

Figure 6: SCOPE's High-Level Architecture

planner that includes the initial list of plan actions, ordering constraints, causal-links, and agenda as input arguments.[4] The last argument corresponds to the output plan solution. All subsequent planning procedure calls are included in the proof structure. Proofs of training problems are used to extract examples of correct and incorrect refinement-selection decisions. Specifically, a "selection decision" for a particular refinement is a planning subgoal to which that refinement was applied. A correct decision is an application of that refinement found on a solution path. An incorrect decision is a refinement application that was tried and subsequently backtracked over. As an example, consider the planning problem that was introduced in Figure 5. The planning subgoal represented by this figure corresponds to the `select-operator` procedure call shown in Figure 7. A stylized version of this subgoal is shown below.

For $\mathcal{S}$ = (0:Start,G:Goal),
$\mathcal{O}$ = (0 < G),
$\mathcal{L}$ = $\emptyset$,
`agenda` = (*Clear(B)*,G),(*On-Table(B)*,G),
Select a new-operator ?OP **to establish** `goal`(*Clear(B)*,G)

This subgoal would be identified as a positive selection decision example for refinement candidate 2 from Figure 5 (adding `Putdown(A)`), and would also be classified as a negative selection decision example for candidates 1 (adding `Unstack(B,A)`). Selection decisions such

[4]Binding constraints in our system are maintained through Prolog, therefore, the set of binding constraints, B, is not explicitly represented in planning procedure calls.

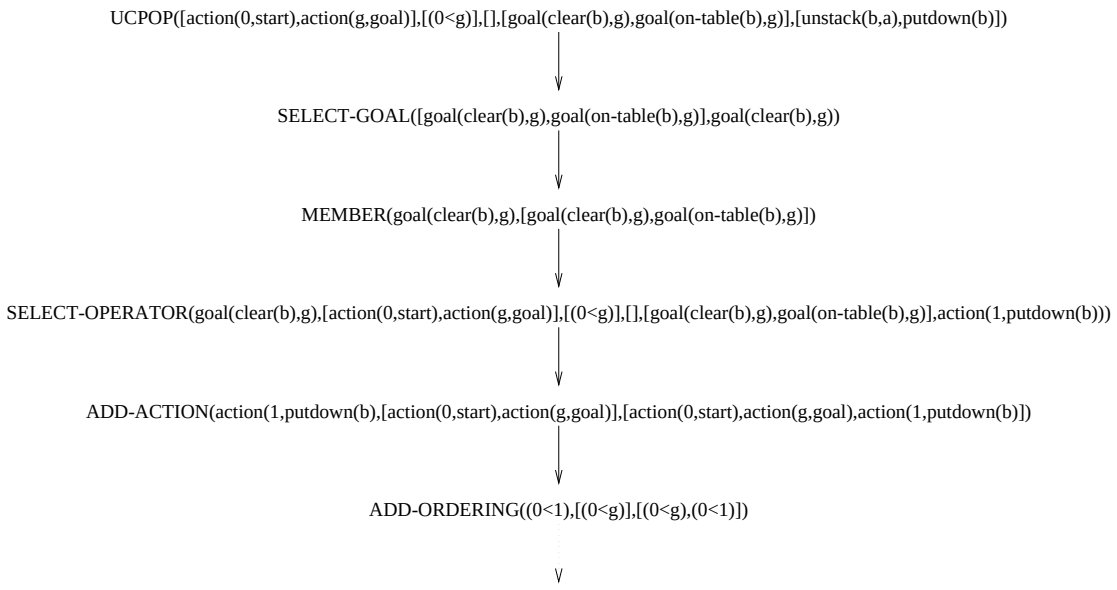

Figure 7: Solution Proof of Planning Problem from Figure 5[5]

as this one are collected for all competing planning refinements. Refinements are considered "competing" if they can be applied in identical planning decisions, such as the three refinement candidates shown in Figure 5. Any given training example may produce numerous positive and negative examples of refinement selection decisions. Selection decision examples are used later in induction to represent positive and negative examples of when to apply particular planning refinements.

The second output of the example analysis phase is a set of generalized proof trees. Standard EBG techniques (Mitchell et al., 1986; DeJong & Mooney, 1986) are used to generalize each training example proof. The goal of this generalization is to remove proof elements that are dependent on the specific example facts while maintaining the overall proof structure. Generalized proof information is used later to explain new planning situations. The top part of an example generalized proof tree is shown in Figure 8. This proof was extracted from the solution trace shown in Figure 7. The generalized proof of an example provides a context which "explains" the success of correct decisions.

Even after EBG has been applied, some tree nodes still contain complex lists as arguments. These arguments correspond to items such as the list of plan actions ($\mathcal{S}$), the list of plan ordering constraints ($\mathcal{O}$), etc. In order to promote more general control knowledge, any lists remaining in a generalized proof tree are generalized directly to variables. Without this extra generalization, proof tree information containing these lists is too specialized to be useful in control rules. This procedure allows SCOPE to build simpler, more general control rules while still retaining enough information in the generalized proof nodes to effectively focus the inductive search.

[5]For space purposes, some information is not shown here. For instance, action data structures should also contain precondition and effect lists.

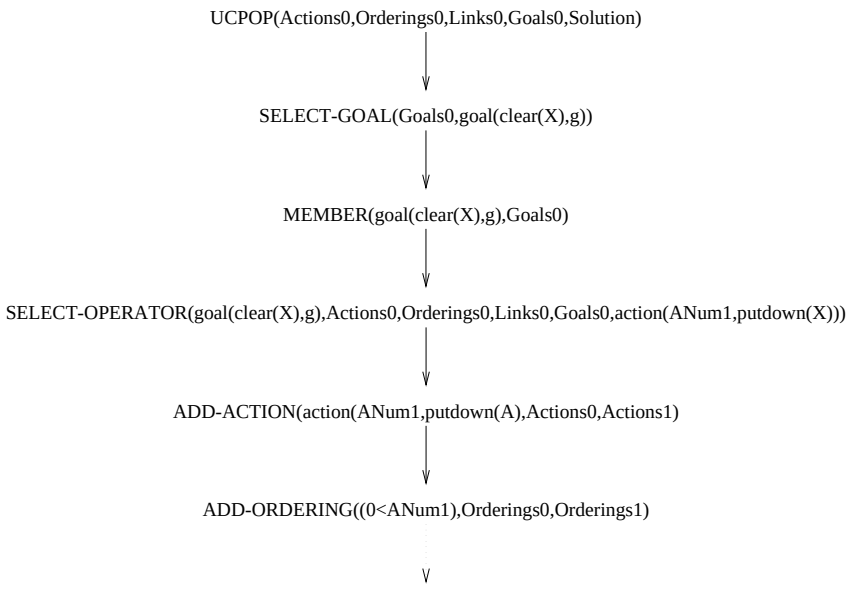

Figure 8: Generalized Proof Tree of Solution Trace from Figure 7

4.2 Control Rule Induction

The goal of the induction phase is to produce an operational definition of when it is useful to apply a plan refinement candidate. Given a candidate, C, we desire a definition of the concept "subgoals for which C is useful". In the blocksworld domain, such a definition is learned for each of the refinements shown in Figure 5. In this context, control rule learning can be viewed as relational concept learning. A number of systems (Quinlan, 1990; Muggleton, 1992; Zelle & Mooney, 1994a) have been designed to tackle this type of learning problem. SCOPE employs a version of Quinlan's FOIL algorithm to learn control rules through induction.

The choice of a FOIL-like framework is motivated by a number of factors. First, the basic FOIL algorithm is relatively easy to implement and has proven efficient in a number of domains. Second, FOIL has a "most general" bias which tends to produce simple definitions. Such a bias is important for learning rules with a low match cost, which helps avoid the utility problem. Third, it is relatively easy to bias FOIL with prior knowledge (Pazzani & Kibler, 1992). In our case, we can utilize the information contained in the generalized proof trees of planning solution traces.

4.2.1 FOIL Algorithm

FOIL attempts to learn a concept definition in terms of a given set of background predicates. This definition is composed of a set of Horn clauses that cover all of the positive examples of a concept, and none of the negative examples. The selection-decision examples collected in the example analysis phase provide the sets of positive and negative examples for each planning refinement candidate.

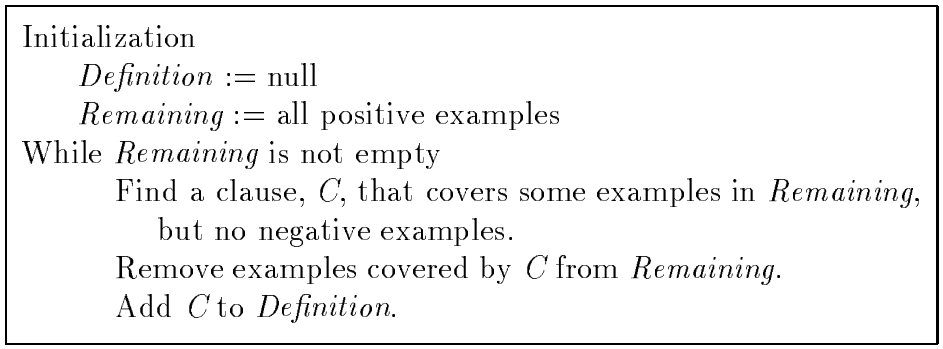

Figure 9: Basic FOIL Covering Algorithm

FOIL may be viewed as a simple covering algorithm which has the basic form shown in Figure 9. The "find a clause" step is implemented by a general-to-specific hill-climbing search. FOIL adds antecedents to the developing clause one at a time. At each step FOIL evaluates all possible literals that might be added and selects the one which maximizes an information-based gain heuristic. This heuristic prefers literals that cover more positive examples and fewer negative examples.

The generation of candidate literals to add to a developing clause normally consists of trying each background predicate with all possible combinations of variables currently in the clause and any new predicate variables. Any predicates that can be used as rule antecedents must be introduced as background knowledge. SCOPE uses an intensional version of FOIL where background predicates can be defined as Prolog programs instead of requiring an extensional representation (Mooney & Califf, 1995).

One major drawback to FOIL (and other similar inductive algorithms) is that the hill-climbing search for a good antecedent can easily explode, especially when there are numerous background predicates with large numbers of arguments. When selecting each new clause antecedent, FOIL tries *all* possible variable combinations for *all* predicates before making its choice. This search grows *exponentially* as the number of predicate arguments increases. SCOPE circumvents this search problem by using the generalized proofs of training examples. By examining the proof trees, SCOPE identifies a small set of potential literals that could be added as antecedents to the current clause definition. Literals are added in a way that utilizes variable connections already established in the proof tree. This approach nicely focuses the FOIL search by only considering literals (and variable combinations) that were found useful in solving the training examples.

4.2.2 Building Control Rules from Proof Trees

The generalized proofs of training examples can be seen as giving the context for the appropriate applications of refinement candidates within a proof. Some nodes of a generalized proof tree contain calls to "operational" predicates. These are usually low-level predicates that have been classified as "easy to evaluate" within the problem domain, and thus can be used to build efficient concept definitions. The operational nodes of a proof represent all of the primitive conditions that had to be satisfied for the proof to succeed. SCOPE employs

induction in an attempt to identify a small set of these simple tests that will provide necessary guidance in determining whether the application of a refinement candidate is likely to lead to a solution. Since test conditions that verify a planning decision are sometimes not executed until much later, it is important to consider an entire example proof instead of just the surrounding context of a particular decision. For instance, the planner might not verify that choosing the action `Putdown(a)` to establish the goal *Clear(a)* is correct until much later in the planning process when it checks to see if some other action has asserted *Holding(a)*.

SCOPE employs the same general covering algorithm as FOIL but modifies the clause construction step. Clauses are successively specialized by considering how their target refinements were used in solving training examples. Suppose we are learning a definition for when each of the refinement candidates in Figure 5 should be applied. The program predicate representing this type of refinement is `select-operator`. This predicate is defined with several arguments including the unachieved goal and an output argument for the selected operator. (Plan state information, such as the list of current plan steps, is also automatically included as arguments to any refinement predicate.) The full predicate head of this refinement is shown below.

```
select-operator(Goal,Steps,Orderings,Links,Agenda,ReturnOp)
```

For each refinement candidate, SCOPE begins with the most general definition possible. For instance, the most general definition covering candidate 1's selection examples is the following; call this clause C.

```
select-operator(Goal,Steps,Orderings,Links,Agenda,unstack(A,B)) :-
    TRUE
```

This overly general definition covers *all* positive examples and *all* negative examples of when to apply candidate 1, since it will always evaluate to true. C can be specialized by adding antecedents to its body. This is done by unifying C's head with a (generalized) proof subgoal that was solved by applying candidate 1 and then adding an operational literal from the same proof tree which shares some variables with the subgoal. For example, one possible specialization of the above clause is shown below.

```
select-operator((clear(B),S1),Steps0,Orderings,Links,Agenda,unstack(A,B)) :-
    establishes(on(A,B),Steps1,S2).
```

Here, a proof tree literal has been added which checks if there is an existing plan step, `S2`, that establishes the goal *On(A,B)*.

Variables in a newly added antecedent can be connected with the existing rule head in several ways. First, by unifying a rule head with a generalized subgoal, variables in the rule head become unified with variables existing in a proof tree. All operational literals in that proof that share variables with the generalized subgoal are tested as possible antecedents. This method initially establishes many relevant variable connections between a rule head and its antecedents.

A second way variable connections may be established is through the standard FOIL technique of unifying variables of the same type. When SCOPE tests a literal for use in a control rule, the literal may contain input parameters that are not bound by the rule head or other existing literals in the rule. If such parameters exist, SCOPE attempts to unify these parameters with terms of the same type that are already present in the rule. For example, the rule shown above has an antecedent with an unbound input, `Steps1`, which does not match any other variables in the rule. SCOPE can modify the rule, as shown below, so that the `Steps1` is unified with a term of the same type from the rule head, `Steps0`.

```
select-operator((clear(B),S1),Steps,Orderings,Links,Agenda,unstack(A,B)) :-
    establishes(on(A,B),Steps,S2).
```

For each unbound input parameter, all possible variable unifications are tested as possible specializations of the current rule and the specialization which maximizes FOIL's information-gain heuristic is selected.

4.2.3 Types of Literals

SCOPE considers several different types of control rule antecedents during induction. Besides pulling literals directly from the generalized proof trees, SCOPE can also use negated proof literals, determinate literals, and literals representing non-codesignation constraints.

A good reason for not selecting one refinement candidate is that another refinement is preferable; therefore, a good control-rule antecedent for one candidate's control rule can often be successfully used as a negated rule antecedent for a competing refinement. Potential *negated* antecedents for a refinement's control rules are determined by combining the sets of possible antecedents for the control rules of all other competing refinements. Negated antecedents can be added to a rule in several ways. Standard FOIL only adds antecedents to a rule until all negative examples are removed. If there are any positives left to cover, a new rule is created. Alternatively, SCOPE can consider adding negated antecedents in order to cover more positives. Instead of only appending negated antecedents to the end of a rule, our induction algorithm considers conjunctively grouping them with existing negated antecedents. This procedure can increase the number of positive examples covered by a rule. For example, assume the induction algorithm is currently considering adding the antecedent `not(ant4)` to the following rule.

```
rulehead :- ant1, not(ant2), ant3.
```

SCOPE can form either of the two rules shown below, where in standard FOIL, only the first rule would have been considered.

```
rulehead :- ant, not(ant2), ant3, not(ant4).
rulehead :- ant, not(ant2, ant4), ant3.
```

The first rule is more specific than the original rule and could possibly exclude more negative examples. The second rule, however, is more general than the original rule and could cover more positive examples.

Determinate literals can be used to introduce new variables into a clause (Quinlan, 1991; Muggleton, 1992). These are literals which produce only one possible binding for each output parameter, and thus their inclusion does not significantly increase the inductive search space. Determinate literals are typically not added through standard induction since they produce little or no gain. However, they can still be useful as rule antecedents by introducing relevant information into a rule. In our induction procedure, all possible determinate literals are automatically added when a new clause is created. If any prove unnecessary they are simply pruned after clause construction has ended. SCOPE currently considers adding two types of determinate literals, find-initial-state(Steps,InitState) and find-final-state(Steps,FinalState); both input a list of plan steps and output the list of propositions representing the initial or final state. These literals are included so control rules can easily access information about the initial and goal states.

The last type of potential literal is a non-codesignation constraint of the form $X_i \neq X_j$, where X_i and X_j are variables existing in a clause. This antecedent checks if two variables (of the same type) are nonunifiable.[6] For example, it may be beneficial to check if two actions are not the same, as in the following rule for candidate 2.

select-operator((S1,clear(A)),Steps0,Orderings,Links,Agenda,putdown(A)) :-
    establishes(holding(A),Steps0,S2),
    S1 $\neq$ S2.

This rule states that `Putdown(A)` will be selected to achieve *Clear(A)* if there is an existing action, `S2`, that establishes *Holding(A)* and `S2` is not the same action which requires `Clear(A)`. Incorporating different antecedent types helps SCOPE to learn expressive control rules that can describe partial-order planning situations.

4.2.4 Relational Clichés

One other feature has been added to SCOPE's algorithm to promote the learning of better control rules. Often during induction, an individual literal may not provide any gain when tested as a possible rule antecedent. However, when grouped with another related literal, the conjunction of the two literals may provide significant gain. In standard hill-climbing, this useful combination of literals may never be discovered if neither literal provides any gain individually. Furthermore, searching through all possible combinations of literals is not a practical consideration. To address this problem, Silverstein and Pazzani (1991) introduced *relational clichés* to suggest potentially useful *combinations* of predicates during relational learning. These clichés provides an efficient means of searching through a restricted subset of the space of predicate combinations.

[6] Literals of the form $X_i = X_j$ are automatically considered through the Prolog unification of rule terms.

Relational clichés consist of two parts:

1. A pattern, which is an abstract description of a conjunction of predicates.
2. A set of restrictions, which constrain the predicates and variable bindings that can be used to fill the associated pattern.

Currently, only the two patterns listed below have been added to our system.

Pattern1: member(A,C), member(B,C)
Restrictions: C must be a list of initial or goal state propositions.

Pattern2: not(member(A,C)),not(member(B,C))
Restrictions: same as above

These two patterns allow induction to test for concepts present in the initial and final states that may be defined by two conditions. These particular patterns were introduced based on our observations of SCOPE's behavior on a transportation domain (introduced in Section 5). In this domain, it is often useful to check whether an object is in one city in the initial state and a different city in the goal state. For instance, if the current goal is to move an object `X` to a certain location `Y`, then, the predicates `member(at-obj(X,Z),InitState)` and `member(same-city(Y,Z),InitState)` could be used to determine whether the object `X` must be transported between cities. To represent this concept, the two general patterns listed above are used. These patterns could be made more specific by adding more specialized restrictions, however, by keeping their description general other useful concepts contained in the initial and goal states may be discovered. Also, these two patterns are not forced to be domain-specific and thus could be useful in other domains besides the transportion domain which originally inspired them. More specific patterns may be eventually added when testing on more complex domains.

To use relational clichés in our induction algorithm, additional rule specializations are generated by adding all combinations of predicates that fit one of the above patterns and associated restrictions. These specializations are then added to the pool of all possible rule specializations from which the rule with the highest gain is chosen. Another possiblity is to only consider relational clichés if no single literal is found to have any gain. However, since our inductive search has proved relatively efficient, we currently allow induction to immediately consider relational clichés as possible rule antecedents.

4.3 Program Specialization Phase

Once refinement selection rules have been learned, they are passed to the program specialization phase which adds this control information into the original planner. The basic approach is to guard each refinement candidate with the selection information. This forces a refinement application to fail quickly on subgoals to which the refinement should not be applied.

```
select-op((clear(B),S1),Steps,OrderCons,Links,Agenda,unstack(A,B)) :-
      find-init-state(Steps,Init),
      member(on(A,B),Init),
      not(member((on(B,C),S1),Agenda),member(on-table(B),Init)),!.

select-op((clear(A),G),Steps,OrderCons,Links,Agenda,putdown(A)) :-
      not(member((on(A,B),G),Agenda)),!.

select-op((clear(A),S1),Steps,OrderCons,Links,Agenda,putdown(A)) :-
      member((on-table(A),S2),Agenda),
      not(establishes(on-table(A),S3)).
```

Figure 10: Learned control rules for two refinement candidates

A decision is also made as to whether the control information has made the planner deterministic. If a refinement rule covers no incorrect selection decisions in the induction phase, then it is assumed that the rule is fully accurate and no other refinement candidates will need to be considered. This type of rule is marked as deterministic by adding a Prolog cut (!) after the last rule condition, which will prevent any backtracking over the refinement selection if all rule conditions are true. If a refinement rule could not exclude all incorrect decisions in the previous phase, then the planner is still allowed to backtrack over the selection of that refinement (i.e. no cut is added). This type of rule can still substantially improve planning efficiency by preventing many inappropriate applications of that refinement.

Figure 10 shows several learned selection rules for the first two refinement candidates (from Figure 5). The first rule allows `Unstack(A,B)` to be applied only when `A` is found to be on `B` initially, and `Stack(B,C)` should not be selected instead. The second and third rule allow `Putdown(A)` to be applied only when `A` should be placed on the table and not stacked on another block. Notice, the first and second rules are marked as deterministic and thus, once found to be true, cannot be backtracked upon. The last rule, however, does not contrain a cut and could be backtracked over if necessary.

5 Preliminary Evaluation of SCOPE

5.1 Experimental Design

Two planning domains were used to test the SCOPE learning system. For the first domain, the standard set of blocksworld operators from Nilsson (1980) were used. The logistics transportation domain of Veloso (1992) was adopted for a second set of experiments. In this domain, packages must be delivered to different locations in several cities. Packages are transported between cities by airplane and within a city by truck. In both domains, a test set of 100 independently generated problems was used to evaluate performance. SCOPE was trained on separate example sets of increasing size. Ten trials were run for each training set size, after which results were averaged. Training and test problems were produced for both domains by generating random initial and final states. In blocksworld, problems contained

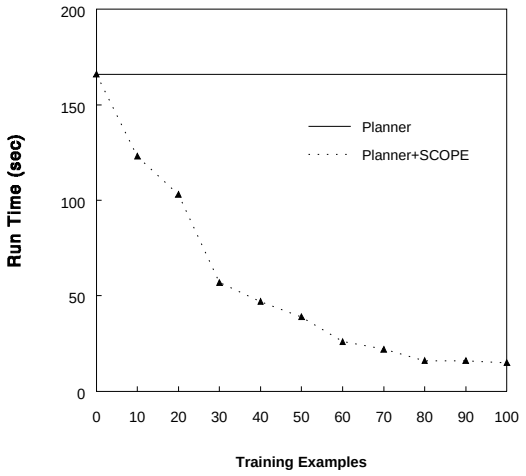

Figure 11: Performance in Blocksworld

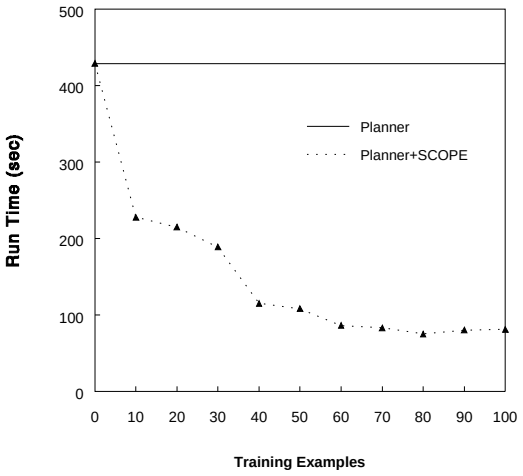

Figure 12: Performance in Logistics

two to six blocks and one to four goals. Logistics problems contained up to two packages and three cities, and one or two goals. No time limit was imposed on planning in either domain, but a uniform depth bound on the plan length was used during testing that allowed for all problems to be solved. All tests were performed on a Sun SPARCstation 5.

For each trial, SCOPE learned control rules from the given training set and produced a modified planner. Since SCOPE only specializes decisions in the original planner, the new planning program is guaranteed to be sound with respect to the original one. Unfortunately, the new planner is not guaranteed to be complete. Some control rules could be too specialized and thus the new planner may not solve all problems solvable by the original planner. In order to guarantee the completeness of the final planner, a strategy used by Cohen (1990) is adopted. If the final planner fails to find a solution to a test problem, the initial planning program is used to solve the problem. When this situation occurs in testing, both the failure time for the new planner and the solution time for the original planner are included in the total solution time for that problem.

5.2 Results

Figures 11 and 12 present the experimental results. The times shown represent the number of seconds required to solve the problems in the test sets after SCOPE was trained on a given number of examples. Each learning curve was generated by averaging the results of 10 trials. In both domains, SCOPE consistently produced a more efficient planner and significantly decreased solution times on the test sets. In the blocksworld, SCOPE produced modified planning programs that were an average of 11.3 times faster than the original planner. For the logistics domain, SCOPE produced programs that were an average of 5.3 times faster. These results indicate that SCOPE can significantly improve the performance of a partial-order planner.

Learned control information was able to cover an average of 97% of the test examples in these domains. On any uncovered examples, the new planner would tend to fail quickly,

System	*Orig. Time*	*Final Time*	*Speedup*	*Orig. %Sol*	*Final %Sol*
UCPOP+EBL	7872	5350	1.47X	51%	69%
SCOPE	5312	1857	2.86X	59%	94%

Table 1: SCOPE vs. UCPOP+EBL

and the original planning program would be used to produce a solution. These failures were usually due to one or two learned rules that were slightly inaccurate. Further extensions to SCOPE should improve the overall accuracy of learned rules and further increase performance results in these domains.

5.3 Comparison to Other Systems

Only one other learning system has been developed to learn search-control rules for a partial-order planner. UCPOP+EBL (Kambhampati et al., 1996) uses standard explanation-based learning to generate control rules in response to planning failures and is discussed in more detail in Section 6. To compare the two systems, we replicated an experiment used by Kambhampati et al. (1996). Problems were randomly generated from a version of the blocksworld domain that contained between three to six blocks and three to four goals.[7] SCOPE was trained on a set of 100 problems. The test set also contained 100 problems and a CPU time limit of 120 seconds was imposed during testing. Data was collected on planner speedup and on the number of test problems that could be solved under the time limit. The results are shown in Table 1. Both systems were able to increase the number of test problems solved, however, SCOPE had a much higher success rate. Overall, SCOPE achieved a better speedup ratio, producing a more efficient planner. By combining EBL with induction, SCOPE was able to learn better planning control heuristics than EBL did alone. These results are particularly significant since UCPOP+EBL requires additional domain axioms which were not provided to SCOPE. It should be noted, however, that since these two systems were run on different platforms, the times reported here represent only a rough comparison.

6 Related Work

6.1 Learning Control for Problem Solving

Early research in learning control rules has been focused on a variety of problem solving applications, such as symbolic integration, eight-puzzle, and the N-Queens problem. To acquire control information, most previous learning systems employed a similar method to SCOPE's of analyzing the search space. Positive and negative examples of problem solver behavior are identified (Mitchell et al., 1983; Langley, 1985), and then control heuristics are learned to cover the positive examples and rule out the negatives. Several early approaches

[7] In order to replicate the experiments of (Kambhampati et al., 1996), the blocksworld domain theory used for these tests slightly differed from the one used for the experiments presented in Section 5. Both domains employed similar predicates however the Section 5 domain definition consists of four operators while the domain used here has only two.

also used a combination of induction and EBL to learn control information. The LEX-2 (Mitchell et al., 1983) and MetaLEX (Keller, 1987) systems constructed rules by inducing over complete explanation-based generalizations of problem-solving traces. SCOPE, on the other hand, uses induction to select the most useful pieces of EBG generalizations.

Some work has also been done on learning approximations to EBL rules. The ULS system (Chase et al., 1989) acquired conservative approximations to EBL rules by simply dropping one or two conditions. ULS was very limited in the rules it could generate, unlike SCOPE, which uses an inductive learning mechanism to build rules from scratch. A more closely related system is AxA-EBL (Cohen, 1990), which integrates an induction mechanism to learn approximate EBL rules. AxA-EBL first learns a control rule by applying EBG to the proof of a correct control decision. A pool of candidate control rules is then formed by considering all *k-bound approximations* of this rules, where a k-bounded approximation is formed by dropping k or less rule conditions. AxA-EBL then searches this pool for a small set of rules that maximizes coverages of the positive examples and minimizes coverage of the negative examples. Although quite successful, this system does has several weaknesses. First, the number of k-bounded approximations grows exponentially in k, thus k is limited to very small values. A second problem is that explanations for subgoals only consider the context of that particular subgoal. Often the conditions which cause a particular operator to fail, lie outside the proof of the specific subgoal to which that operator was applied.

The DOLPHIN system (Zelle & Mooney, 1993) improves on AxA-EBL by using a more powerful induction algorithm (FOIL) and analyzing proof trees of entire problems instead of only explaining individual subgoal successes. Though DOLPHIN has been shown successful at improving program performance in several domains, as mentioned in Section 4, DOLPHIN had little success improving the performance of a partial-order planner. Previously, DOLPHIN had been mainly applied to simple logic programs whose search space was easy to analyze. However, when tested on our UCPOP planner, DOLPHIN, could not generate rules that were expressive enough to describe partial-order planning situations. DOLPHIN also has difficulty successfully generalizing explanations generated by UCPOP. Generalized proof tree information was too complex to be useful in building effective search-control rules. SCOPE has greatly expanded upon the DOLPHIN algorithm in several ways. First, extra generalization techniques were applied to some proof tree literals so that proof information would be more useful in building control rules. Second, the induction algorithm was significantly extended to include features such as variable matching, additional literals types, and relational clichés.

6.2 Learning Control Specifically for Planning

A significant amount of research in learning control knowledege has been explicitly directed towards improving planning systems. Most of this research has concentrated on linear, state-based planners. For instance, the PRODIGY planning and learning system (Minton, 1989) employs a version of EBL to learn control rules for a linear, state-based planner. Rules are learned that can select, reject, or prefer refinement candidates for several different decision types. A number of other systems have also applied EBL to learn search-control for planning. STATIC (Etzioni, 1993) acquires control rules by analyzing the problem domain theory. This system uses EBL to analyze a graph structure that captures the precondition/effect depen-

dencies between the actions in the domain. This analysis is then used to derive goal-ordering rules for the PRODIGY state-based planner. FAILSAFE (Bhatnagar & Mostow, 1994) was designed to learn control rules in domains where the underlying domain theory was recursive. FAILSAFE uses a forward-searching state-based planner and learns by building incomplete explanations of its execution time failures.

Not all learning approaches have relied on EBL. GRASSHOPPER (Leckie & Zuckerman, 1993) uses an inductive approach to learn planning control knowledge. Given a set of training examples, GRASSHOPPER looks for sets of similar decisions that could form the basis for search-control rules. Rule preconditions are generated by generalizing the current world state information that held at the beginning of each decision.

All of these learning systems have been shown successful at improving planner performance in a variety of domains. However, unlike SCOPE, each system's architecture is limited to apply only to a linear, state-based planner. For instance, most of the rules generated by these early systems heavily rely on current state information which is not available in a partial-order planner. These systems would also have difficulty analyzing the search space of a partial-order planner. Since partial-order planners operate in a plan-space instead of a state-space, the process of creating and generalizing explanations is more complex. Removing the linearity assumption adds further complications for learning search-control not addressed by these early systems, since many more search paths must be considered. Unfortunately, very few systems have been built to acquire control knowledge for more modern planning techniques.

HAMLET (Borrajo & Veloso, 1994b) is one more recent system, which learns control knowledge for the nonlinear planner underlying PRODIGY4.0. HAMLET uses a combination of EBL with induction to acquire control rules. HAMLET first generates a bounded explanation of each planning decision and then incrementally refines any incomplete or inaccurate explanations. Control rules can either be specialized if they are found to cover negative examples or generalized if they are found to exclude positive examples. Unlike SCOPE, HAMLET uses EBL to build rules, and induction is primarily used to refine learned knowledge. Also, though the Prodigy4.0 planner is fairly modern, it is still considered a state-based planner and many rule conditions used by HAMLET are directly dependent on current state information. It is thus unclear, how HAMLET would perform on a partial-order (or plan-space) planner. HAMLET has successfully improved the performance of the PRODIGY4.0 planner in the blocksworld and logistics planning domains. When comparing to the results reported in (Borrajo & Veloso, 1994b), SCOPE achieves a greater speedup factor in blocksworld (11.3 vs 1.8) and in the logistics domain (5.3 vs 1.8). However, since different problem distributions were used in these experiments and HAMLET is built upon a different planning platform, we can only draw a rough comparison between HAMLET and SCOPE.

The only system besides SCOPE to learn control information for partial-order planning is UCPOP+EBL (Kambhampati et al., 1996). This system also learns search control rules for UCPOP, but uses a purely explanation-based approach. Specifically, UCPOP+EBL employs the standard EBL techniques of regression, explanation propagation and rule generation to acquire search-control rules. Rules are learned only in response to past planning failures. UCPOP+EBL learns from both analytical failures (dead-end search paths) and depth limit failures (the search path crosses a depth limit). In order to explain the failures of search

paths that cross over a depth limit, this system can utilize extra domain axioms. These axioms, which are defined as "readily available physical laws of the domain", help detect and explain inconsistencies at some depth limit failures. UCPOP+EBL is limited in the rules it can learn, however, since only explainable failures can be utilized for learning. Even with additional domain axioms, it may be difficult or impossible to explain some depth limit failures. SCOPE, on the other hand, can learn rules to avoid all unpromising paths, as long as a solution path has been identified. SCOPE also requires no additional domain information, such as the domain axioms provided to UCPOP+EBL.

UCPOP+EBL has been shown to improve planning performance in several domains, including the blocksworld, however, as shown in Section 5, SCOPE outperforms UCPOP+EBL in this domain. Unfortunately, neither systems appears to generate the optimal set of control rules since both fail to solve all test examples. An interesting future research idea is to combine techniques from both systems to in order to learn more effective control knowledge. This research propspect is discussed further in the following section.

7 Future Work

There are several issues we hope to address in future research. First, several enhancements to SCOPE's algorithm are planned. These include incorporating a method that directly evaluates control-rule utility, and including a method of constructive induction which could further rule quality. Second, SCOPE should be tested on more complex domains that contain conditional effects, universal quantification, and other more-expressive planning constructs. Next, we want to to examine ways of using SCOPE to improve plan quality as well as planner efficiency. Finally, we hope to apply the SCOPE learning system to a different type of planning algorithm. Each of these topics is addressed below.

7.1 Enhancing SCOPE

Currently, SCOPE automatically includes any learned rule in its final set of control rules, however, some rules are probably much more useful than others. The utility of individual rules can often dramatically vary and too many rules of low utility can even lead to lower performance (Minton, 1988). This occurence, commonly known as the *utility problem*, can be prevented by only including the most useful control rules in the final planner. Thus, we would like to incorporate a method into SCOPE that directly evaluates control-rule utility. Researchers have introduced a variety of techniques for determining the best rules to save (Greiner & Likuski, 1989; Markovitch & Scott, 1989; Subramanian & Feldman, 1990; Gratch & DeJong, 1992). As yet, no one has applied such techniques to evalutate rules for a partial-order planner, however, we feel that such a method could be easily integrated into our learning system.

Another possible improvement is to replace or modify the standard FOIL information-gain heuristic currently used by SCOPE's induction algorithm. Though the results with this heuristic have been encouraging, experimenting with different heuristics may be beneficial. One possibility is to replace this heuristic with a metric that more directly measures rule

utility. This modification could improve performance by encouraging the system to only select highly-useful control rule antecedents. Another problem with the current heuristic is that a good rule is often discarded because it covers one or two negative control examples. Even if such a rule is retained, it will be considered "nondeterministic" (no cut will be added) and could be bactracked upon, causing a potential loss in speedup. SCOPE could benefit from methods for handling noisy data that have been in employed in FOIL and other related systems (Quinlan, 1990; Muggleton, 1992; Lavrač & Džeroski, 1994). In SCOPE's framework, a small percentage of incorrectly covered examples could be treated as noise, thereby allowing some good rules to be retained and more rules to be marked as deterministic. This procedure could cause even more backtracking to be eliminating, resulting in lower solution times for many examples. Problems that are not correctly covered by rules are already handled by retaining a backup of the original planner. Since most backtracking would be eliminated, the new planner should fail quickly on these examples, thus incurring very little extra time to solve them. Speedup gain on all other examples could make this approach beneficial overall.

Another planned enhancement is the incorporation of constructive induction to invent new predicates when needed. Currently, the pool of possible control rule antecedents is drawn mainly from program predicates which were used in the main planning algorithm. However, for complete optimization of a planner or other problem solver, it is often necessary to introduce new concepts for use in the control language. For instance, it may be helpful in the logistics domain, to have a predicate which tests whether two objects are initially in the same city. Currently, it is possible for the user to provide SCOPE with extra concepts that could be found useful in the form of determiniate literals or relational cliches. However, a better method would be for the system to learn these concepts automatically. Constructive induction techniques have been implemented in a several ILP systems (Kijsirikul et al., 1992; Zelle & Mooney, 1994b) and could help SCOPE learn more accurate and efficient conrol rules.

Furthermore, it is often the case that concepts that are useful in making a control decision about a certain plan refinement are also useful for making decisions about other related refinements. However, even with constructive induction, these "shared" concepts must be relearned from scratch for each new control rule. In a FOIL-like inductive learner, it is relatively easy to make a newly constructed concept available for reuse to simple adding it to the list of predicates that can be used as control rule antecedents. Unfortunately, the definition of this concept may be incomplete or incorrect, depending on the particular set of training examples that were used to construct it. This problem can be remedied by the following approach. If a previously defined concept is found to be the best choice for the next control-rule antecedent, yet it does not cover all the positive control examples, this concept can be redefined by merging the original set of control examples with the new set and then recursively inducing rules to cover all positive examples. This type of *concept sharing* procedure could help further reduce the inductive search space, and make SCOPE's learning algorithm more efficient.

One last possible research direction is to utilize information about planning failures to learn more effective control rules. Currently, SCOPE only uses the generalized solution proofs of the training examples to bias the inductive search. Another approach is to also utilize explanations of why particular search paths failed. For instance, the generalized failure explanations created by the UCPOP+EBL learning system could also be used to bias the

inductive search. This approach would allow control rules to directly utilize relelvant failure information. In particular, rules would have access to the sets of conditions which caused many search paths explored in the training examples to fail. This information could help SCOPE to build more effective rules which more accurately avoid unpromising search paths.

7.2 Experimental Evalution

In order to evaluate the efficacy of the SCOPE framework for improving planning efficiency, experimental tests on larger problems are planned. The experimental methodology will involve producing learning curves likes those shown in Section 5 which demonstrate performance improvement. We also plan to present more extensive comparisons between our approach and any other competing methods for partial-order planning such as UCPOP+EBL.

SCOPE will be tested on more complex domains that contain expressive constructs such as conditional effects and universal quantification. Several domains have already been identified that would provide an interesting testbed for future experiments. The first is the UM Translog domain being developed at the University of Maryland. UM Translog (Andrews et al., 1995) was inspired by the logistics transportation domain used in Section 5, however, it is an order of magnitue larger in size (41 actions vs. 6) and provides more complex features and goal interactions. In this domain, packages must still be delivered between cities, however there are more modes of transportation, vehicles and packages have special loading methods, and some routes of transportation can be temporarily unavailable. The detailed set of operators in this domain provide for long plans (an average of 40 steps) with many possible solutions to the same problem.

The PRODIGY Process Planning domain (Gil, 1991) is another large-scale complex domain that could be used to further evaluate SCOPE's learning algorithm. This domain incorporates detailed knowledge about the automation of manufacturing processes and is one of the largest domains available for general-purpose planners. Actions include activites such as machining, joining and finishing of parts. In total, this domain contains 81 different operators and plans often consist of over 100 steps. Both the UM Translog and the Process Planning domains provide an extensive set of interesting entities and actions, which can be used to specify complex planning problems, and thus would provide very interesting test domains for SCOPE and other planning and learning systems.

7.3 Improving Plan Quality

We also plan to examine ways of using SCOPE to improve plan quality as well as planner efficiency. Borrajo and Veloso (1994a) and Pérez and Carbonell (1994) have used learned control information to guide the PRODIGY4.0 planner towards better solutions. In this type of approach, rules are acquired that will improve the final plan as opposed to just improving planner efficiency. There are a variety of notions of what constitutes a "good plan". The most common metrics include the length of a plan, the plan's total execution time, and the number of resources required to execute the plan. SCOPE could be modified to collect positive control examples only from high-quality solutions so that control rules are focused on one

or more quality issues as well as speedup. To date, little work has been done on improving plan quality in a partial-order planner; and no work has been attempted on *learning* control knowledge for improving plan quality for such a planner. Thus, we feel this is an important line of research that would be beneficial to the planning community.

7.4 Applying SCOPE to Other Planning Systems

Finally, we want to demonstrate that SCOPE's learning algorithm is not limited to improving the performance of only one type of planner. We intend to apply SCOPE to at least one different planning algorithm in hopes of achieving similar performance gains. There are several other types of planners besides partial-order that are prominent in the planning community today. One is an *hierarchical-task network* (HTN) planner (Erol et al., 1994a). As opposed to most *operator-based* planners, HTN planners specify plan modifications in terms of task reduction rules. These reduction rules are then used to decompose abstract goals into lower level tasks. A set of similar constraints as found in an operator-based planner (e.g. orderings, causal-links) is maintained in an HTN planner and many of the same methods can be used to resolve possible conflicts. HTN planners have several decision points where control knowledge could be useful, including what task reduction rule to apply, and what constraint resolution method to use. HTN planners are argued by some researchers to be more useful for real-world applications since they offer more flexibility in expressing domain knowledge. There are several well-developed HTN algorithms that would be good testbeds for SCOPE. These include UMCP (Erol et al., 1994b), developed at the University of Maryland, and COLLAGE (Lansky & Getoor, 1995), developed at the NASA Ames Research Center.

Another breed of planners that we will consider are those that use a combination of *plan-space* and *state-space* techniques. Kambhampati and Srivastava (1995) recently introduced the UCP planning algorithm which casts plan-space and state-space planning methods into a single framework. UCP has the freedom to interleave these two refinement strategies on a singular plan represenation. Veloso and Stone (1995) has also developed a new approach to planning that combines these two refinement strategies by using a flexible approach to ordering commitments. FLECS can use both least-commitment and eager-commitment strategies and can vary its use across different problems and domains. This strategy integrates techniques from both partial-order and total-order planners. Both UCP and FLECS could highly benefit from learned control knowledge since they have been especially designed to take advantage of heuristic knowledge at the choice points just described.

8 Conclusion

SCOPE provides a new mechanism for learning search-control information in planning systems. By using a unique combination of explanation-based learning and inductive logic programming, SCOPE can acquire high-utility control rules for a complex planning system. Specifically, rules are learned by inducing concept definitions of when to apply particular plan refinements. Explanation-based generalization aids the inductive search by focusing it

towards the best pieces of background information. Unlike most approaches which are limited to linear, state-space planners, SCOPE learns control rules for the newer, more effective partial-order planners. When evaluated on a version of the UCPOP planning algorithm, SCOPE significantly improved planner performance in both the blocksworld and logistics transportation domains. SCOPE also outperforms a the competing method based only on EBL. Future enhancements to SCOPE include incorporating a method for rule utility analysis and utilizing contructive induction techniques to learn more effective control rules. SCOPE will also be evaluated on more complex planning domains; future experiments include measuring planner speedup and plan quality improvements. Finally, SCOPE will be applied to a new planning system to demonstrate the versatility of SCOPE's control rule learning algorithm.

References

Andrews, S., Kettler, B., Erol, K., & Hendler, J. (1995). UM Translog: A planning domain for the development and benchmarking of planning systems. Tech. rep. CS-TR-3487, Institute for Advanced Computer Studies, University of Maryland.

Barrett, A., & Weld, D. (1994). Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, *67*, 71–112.

Barrett, A., Golden, K., Penberthy, S., & Weld, D. (1993). UCPOP: User's manual (version 2.0). Tech. rep. 93-09-06, Department of Computer Science and Engineering, University of Washington.

Bhatnagar, N., & Mostow, J. (1994). On-line learning from search failure. *Machine Learning*, *15*, 69–117.

Borrajo, D., & Veloso, M. (1994a). Incremental learning of control knowledge for improvement of planning efficiency and plan quality. In *AAAI-94 Fall Symposium on Planning and Learning*, pp. 5–9 New Orleans, LA.

Borrajo, D., & Veloso, M. (1994b). Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML-94*, pp. 64–82 Springer Verlag.

Chase, M., Zweben, M., Piazza, R., Burger, J., Maglio, P., & Hirsh, H. (1989). Approximating learned search control knowledge. In *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 40–42 Ithaca, NY.

Chien, S. (1989). Using and refining simplifications: Explanation-based learning of plans in intractable domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* Detroit, MI.

Chien, S., & DeJong, G. (1994). Incremental reasoning in explanation-based learning of plans. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago.

Chien, S., Hill, R., Wang, X., Estlin, T., Fayyad, K., & Mortensen, H. (1995). Why real-world planning is difficult: A tale of two applications. In *Proceedings of the Third European Workshop on Planning*, pp. 305–317 Assisi Italy.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 268–276 Austin, TX.

DeJong, G. F., & Mooney, R. J. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*(2), 145–176. Reprinted in *Readings in Machine Learning*, J. W. Shavlik and T. G. Dietterich (eds.), Morgan Kaufman, San Mateo, CA, 1990.

Erol, K., Nau, D., & Hendler, J. (1994a). HTN planning: Complexity and expressivity. In *Proceedings of the Eleventh National Conference on Ariticial Intelligence* Seattle.

Erol, K., Nau, D., & Hendler, J. (1994b). UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago.

Etzioni, O. (1993). Acquiring search control knowledge via static analysis. *Artificial Intelligence*, *60*(2).

Fargher, H., & Smith, R. (1994). Planning in a flexible semiconductor manufacturing environment. In Zweben, M., & Fox, M. (Eds.), *Intelligent Scheduling*, pp. 545–580. Morgan Kaufmann, San Francisco, CA.

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3/4).

Gil, Y. (1991). A specification of manufacturing processes for planning. Tech. rep. CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Gratch, J., & DeJong, G. (1992). COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 235–240 San Jose, CA.

Greiner, R., & Likuski, J. (1989). Incorporating redundant learned rules: A preliminary formal analysis of ebl. In *Proceedings of the Eleventh International Conference on Ariticial Intelligence*, pp. 744–749 Detroit, MI.

Kambhampati, S., & Chen, J. (1993). Relative utility of EBG based plan reuse in partial ordering vs. total ordering. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 514–519 Washington, D.C.

Kambhampati, S., Katukam, S., & Qu, Y. (1996). Failure driven search control for partial order planners: An explanation based approach. *Artificial Intelligence*, Forthcoming.

Kambhampati, S., & Srivastava, B. (1995). Universal classical planner: An algorithm for unifying state-space and plan-space planning. In *Proceedings of the Third European Workshop on Planning*, pp. 81–94 Assisi Italy.

Katukam, S., & Kambhampati, S. (1994). Learning explanation-based search control for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 582–587 Seattle,WA.

Keller, R. (1987). *The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance*. Ph.D. thesis, Rutgers University, New Brunswick, N. Also appears as tech. report ML-TR-7.

Kijsirikul, B., Numao, M., & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 44–49 San Jose, CA.

Langley, P. (1985). Learning to search: From weak methods to domain specific heuristics. *Cognitive Science*, *9*(2), 217–260.

Langley, P., & Allen, J. (1991). The acquisition of human planning expertise. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 80–84 Evanston,IL.

Lansky, A., & Getoor, L. (1995). Scope and abstraction: Two criteria for localized planning. In *Proceedings of the Fourteenth International Conference on Ariticial Intelligence*, pp. 1612–1618 Montreal, CA.

Lavrač, N., & Džeroski, S. (Eds.). (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Leckie, C., & Zuckerman, I. (1993). An inductive approach to learning search control rules for planning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1100–1105 Chamberry,France.

Markovitch, S., & Scott, P. D. (1989). Utilization filtering: A method for reducing the inherent harmfulness of deductively learning knowledge. In *Proceedings of the Eleventh International Conference on Ariticial Intelligence*, pp. 738–743 Detroit, MI.

McDermott, D. (1991). Regression planning. *International Journal of Intelligent Systems*, *6*, 357–416.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 564–569 St. Paul, MN.

Minton, S. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Minton, S., Drummond, M., Bresina, J. L., & Phillips, A. B. (1992). Total order vs. partial order planning: Factors influencing performance. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 83–92 Cambridge,CA.

Mitchell, T., Utgoff, T., & Banerji, R. (1983). Learning problem solving heuristics by experimentation. In Michalski, R., Mitchell, T., & Carbonell, J. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Palo Alto, CA.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*(1), 47–80.

Mooney, R. J., & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, *3*, 1–24.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339–352 Ann Arbor, MI.

Muggleton, S., King, R., & Sternberg, M. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, *5*(7), 647–657.

Muggleton, S. H. (Ed.). (1992). *Inductive Logic Programming*. Academic Press, New York, NY.

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

Pazzani, M., & Kibler, D. (1992). The utility of background knowledge in inductive learning. *Machine Learning*, *9*, 57–94.

Penberthy, J., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 113–114 Cambridge,MA.

Pendault, E. (1989). ADL: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*.

Pérez, M. A., & Carbonell, J. (1994). Control knowledge to improve the plan quality. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago, IL.

Porter, B. W., & Kibler, D. F. (1986). Experimental goal regression: A method for learning problem-solving. *Machine Learning*, *1*(3), 249–285.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Los Altos, CA.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning* Evanston, IL.

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pp. 3–20 Vienna.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Silverstein, G., & Pazzani, M. J. (1991). Relational clichés: Constraining constructive induction during relational learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 203–207 Evanston, IL.

Subramanian, D., & Feldman, R. (1990). The utility of EBL in recursive domains. In *Proceedings of the Eighth National Conference on Ariticial Intelligence*, pp. 942–949 Boston, MA.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* Detroit, MI.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrated planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence Research*, *7*(1).

Veloso, M., & Stone, P. (1995). FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, *3*, 25–52.

Veloso, M. M. (1992). *Learning by Analogical Reasoning in General Problem Solving.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Warren, D. (1974). WARPLAN: A system for generating plans. Tech. rep. Memo No. 76, Department of Computational Logic, University of Edinburgh.

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1106–1111 Chambery, France.

Zelle, J. M., & Mooney, R. J. (1994a). Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 343–351 New Brunswick, NJ.

Zelle, J. M., & Mooney, R. J. (1994b). Inducing deterministic Prolog parsers from treebanks: A machine learning approach. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 748–753 Seattle, WA.

Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M., & Eskey, M. (1992). Learning to improve constraint-based scheduling. *Artificial Intelligence*, *58(1-3)*, 271–296.