STATIC
A Problem-Space Compiler for PRODIGY

Oren Etzioni
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
etzioni@cs.washington.edu

Abstract

Explanation-Based Learning (EBL) can be used to significantly speed up problem solving. Is there sufficient structure in the definition of a problem space to enable a static analyzer, using EBL-style optimizations, to speed up problem solving without utilizing training examples? If so, will such an analyzer run in reasonable time? This paper demonstrates that for a wide range of problem spaces the answer to both questions is "yes."

The STATIC program speeds up problem solving for the PRODIGY problem solver without utilizing training examples. In Minton's problem spaces [1988], STATIC acquires control knowledge from twenty six to seventy seven times faster, and speeds up PRODIGY up to three times as much as PRODIGY/EBL. This paper presents STATIC's algorithms, derives a condition under which STATIC is guaranteed to achieve polynomial-time problem solving, and contrasts STATIC with PRODIGY/EBL.

Introduction

Prieditis [1988] and van Harmelen & Bundy [1988] have pointed to the affinity between Partial Evaluation (PE) [van Harmelen, 1989] and Explanation-Based Learning (EBL) [Dejong and Mooney, 1986, Mitchell *et al.*, 1986] suggesting that an EBL-like process could be performed without utilizing training examples. These papers raise a number of interesting questions.

The training example given to EBL helps to focus its learning process. Can PE, which does not utilize training examples, be performed in a reasonable amount of time on standard EBL domain theories? van Harmelen & Bundy's illustrative partial evaluator will not terminate on recursive programs. Yet standard EBL theories (e.g., the ones used by PRODIGY/EBL [Minton, 1988]) are recursive. Can PE handle recursion appropriately? EBL is widely used for acquiring search control knowledge [Minton *et al.*, 1989a]. How will control knowledge obtained via PE compare with that acquired by EBL? In the past, PE has been applied to inference tasks. Can PE be extended to analyze the goal interactions (e.g., goal clobbering) found in planning tasks? Finally, what guarantees can we make regarding STATIC's impact on problem-solving time?

This paper addresses these questions by describing STATIC, a problem-space analyzer which utilizes PE, and comparing its performance with that of PRODIGY/EBL—a state-of-the-art EBL system. The following section provides some background on PRODIGY and PRODIGY/EBL. The subsequent sections present STATIC, and derive a condition under which STATIC is guaranteed to achieve polynomial-time problem solving. STATIC's performance is compared with that of PRODIGY/EBL, and STATIC is contrasted with standard partial evaluators and other static analyzers. The paper concludes by considering the lessons learned from the STATIC case study.

The PRODIGY System

PRODIGY is a domain-independent means-ends analysis problem solver [Minton *et al.*, 1989a] that is the substrate for EBL, STATIC, and a variety of learning mechanisms [Carbonell and Gil, 1990, Joseph, 1989, Knoblock, 1991, Knoblock *et al.*, 1991, Veloso and Carbonell, 1990]. A complete description of PRODIGY appears in [Minton *et al.*, 1989b]. The bare essentials follow.

PRODIGY's default search strategy is depth first. The search is carried out by repeatedly executing the following decision cycle: choose a search-tree node to expand, choose a subgoal at the node, choose an operator relevant to achieving the subgoal, and choose bindings for the operator. Control rules can direct PRODIGY's choices at each step in this decision cycle. A sample control rule appears in Table 1. PRODIGY matches the control rule against its current state. If the antecedent of the control rule matches, PRODIGY abides by the recommendation in the consequent.

PRODIGY/EBL

PRODIGY/EBL acquires control rules for PRODIGY by analyzing its problem-solving traces. PRODIGY/EBL's primary target concepts are *success*, *failure*, and *goal interaction*. PRODIGY/EBL finds instances of these concepts in PRODIGY's traces and derives operational suf-

```
(REJECT-UNSTACK
  (if
    (and (current-node N)
         (current-goal N (holding Block))
         (candidate-op N unstack)
         (known N (not (on Block Blck2)))))
  (then (reject operator unstack)))
```

Table 1: A Blocksworld control rule telling PRODIGY to reject the operator UNSTACK when the block to be unstacked is not on any other block.

ficient conditions for the concepts via EBL. These conditions become the antecedents of PRODIGY/EBL's control rules and the target concept determines the recommendation expressed in the consequent of the rule. The rule in Table 1, for example, is formed by analyzing the failure of the Blocksworld operator UNSTACK.

STATIC

STATIC is a program that acquires search control knowledge for PRODIGY by statically analyzing PRODIGY's problem space definitions. Like PRODIGY/EBL, STATIC analyzes success, failure and goal interaction. Unlike PRODIGY/EBL, STATIC only learns from nonrecursive explanations and does not require training examples, utility evaluation, or rule compression.

This paper reviews briefly STATIC's graph representation and algorithms for analyzing failure and success (introduced in [Etzioni, 1990b]) and focuses on STATIC's algorithms for analyzing goal interaction and on a detailed comparison with PRODIGY/EBL.[1] The main new result presented is a comparison of the time required to generate control knowledge via STATIC with the time required to train PRODIGY/EBL. In PRODIGY/EBL's problem spaces, STATIC generates control knowledge from twenty six to seventy seven times faster than PRODIGY/EBL.

STATIC's input is a problem space definition; its output is a set of control rules that are intended to speed up PRODIGY's problem solving. STATIC's operation can be divided into four phases. First, it maps the problem space definition to a set of Problem Space Graphs (PSGs), one for each potential subgoal.[2] Second, it labels each PSG node and computes the conditions under which subgoaling or back-chaining on the node would lead to failure. Third, it creates control rules whose antecedents are composed of these conditions (or their negation). The rules guide PRODIGY's operator and bindings choices. Finally, STATIC searches for goal clobbering and prerequisite violation by analyzing PSG pairs, and forms goal ordering and operator preference rules based on this analysis.

Problem Space Graphs (PSGs)

A PSG is an AND/OR graph that represents the goal/subgoal relationships in a problem space. STATIC constructs PSGs by partially evaluating the problem space definition; PSGs are independent of any state information. This section describes PSGs in more detail. A pseudo-code description of STATIC's algorithm for constructing PSGs appears in Table 2.

A PSG consists of disjoint subgraphs each of which is rooted in a distinct subgoal literal. Each subgraph is derived by symbolically back chaining on the problem space's operators from the root. The root literal is connected via OR-links to all the operators whose effects match the literal, and each operator is connected via AND-links to its preconditions. Thus, the PSG nodes are an alternating sequence of (sub)goals and operators; the PSG edges are either AND-links or OR-links. Figure 1 depicts the Blocksworld PSG subgraph rooted in `(holding V)`. The graph is directed and acyclic. Two operators that share a precondition have AND-links to the same node, so the graph is not a tree.

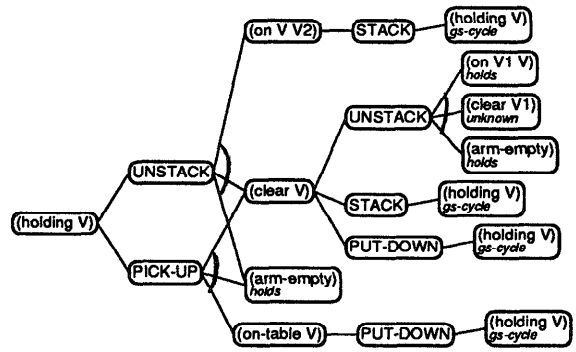

Figure 1: The `holding` PSG.

A successful match between an operator's effects and a subgoal imposes codesignation constraints between the arguments to the subgoal literal, and the operator's variables. The operator's preconditions are partially instantiated to reflect these constraints. For example, since the goal is `(holding V)`, UNSTACK's first precondition is partially instantiated to `(on V V2)`.

The expansion of the graph is terminated under well-defined criteria discussed in [Etzioni, 1990b]. The most important criterion is the termination of PSG expansion whenever predicate repetition (on a path from a node to the root) is detected. This criterion suffices to prove that every PSG is finite [Etzioni, 1990a].

[1] STATIC was introduced (in approximately one page) in [Etzioni, 1990b] to demonstrate PRODIGY/EBL's reliance on nonrecursive problem space structure.

[2] A potential subgoal is an uninstantiated literal found in the effects of an operator. Potential subgoals are enumerated by scanning the problem space definition.

Input: operators, constraints on legal states, and a uninstantiated goal literal g.
Output: A PSG for g (e.g., Table 1).

The goal stack of a node n in the PSG, *goal-stack(n)*, is the unique set of subgoal nodes on a path from n to the root. The function *holds(p,goal-stack(p))* determines whether the precondition p necessarily holds given its goal stack and the constraints on legal states. $ops(g)$ refers to the set of operators that can potentially achieve g.
Algorithm:

1. Create a subgoal node for g.
2. For each operator $o \in ops(g)$:
   (a) Partially instantiate o to o_g.
   (b) Create an operator node for o_g and OR-link it to g.
   (c) If any of o_g's preconditions appears on its goal stack, then create a subgoal node corresponding to that precondition, label it *gs-cycle*, and AND-link it to o_g.
   (d) Else, for each precondition $p \in precs(o_g)$:
      i. If a previously expanded sibling operator shares p, then a node for p already exists. AND-link o_g to the existing node.
      ii. Otherwise, create a new node p and AND-link it to o_g.
      iii. If $holds(p, (\text{goal-stack}(p))$ then label p *holds*.
      iv. If p's predicate appears on o_g's goal stack, then label p *unknown* (i.e. recursion could occur here).
      v. If no operator matches p, then label p *unachievable*.
      vi. Else, return to step 1 with p as the current subgoal g.

Table 2: Constructing a PSG.

Having constructed a set of PSGs corresponding to a problem space, STATIC proceeds to label each internal node in the PSGs with one of the labels *failure*, *success*, or *unknown* indicating the eventual outcome of problem solving when PRODIGY reaches a corresponding node in its search. The labeling is performed using a three-valued "label logic" where the label of an operator node is the conjunction of its preconditions' labels and the label of a precondition node is the disjunction of the labels of the operators whose effects match the precondition. PRODIGY control rules that guide operator and bindings choices are constructed based on this labeling process. See [Etzioni, 1990a] for more details.

Analyzing Goal Interactions in STATIC

Goal interactions such as goal clobbering and prerequisite violation cause PRODIGY to backtrack. STATIC anticipates these goal interactions by analyzing PSG pairs, and generates goal ordering and operator preference control rules that enable PRODIGY to avoid goal interactions. STATIC is *not complete* in that it will not anticipate all possible goal interactions, but it is *correct* in that the interactions that STATIC anticipates would in fact occur, if it were not for STATIC's control rules.

STATIC anticipates goal interactions that will *necessarily* occur. That is, STATIC reports that two goals interact, under certain constraints, only if the goals interact in *every* state that obeys the constraints. An alternative optimization strategy (taken by Knoblock's Alpine [1991], for example) is to anticipate *possible* goal interactions.

While STATIC analyzes both goal clobbering and prerequisite violation, this section considers only goal clobbering. The section begins by formally defining goal clobbering. The section then presents STATIC's algorithm for anticipating goal clobbering by analyzing PSGs. Two important simplifications are made in the presentation for the sake of brevity and clarity. First, the presentation is restricted to propositional goals even though STATIC actually analyzes variablized goals and codesignation constraints between them. Second, although necessary goal interactions are sometimes "conditional" (that is, occur necessarily only when the set of possible states is constrained appropriately) the discussion does not describe how STATIC derives these constraints. See [Etzioni, 1990a] for a complete discussion.

A plan for achieving a goal g is said to *clobber* a protected goal pg when pg is deleted in the process of executing the plan. In Minton's machine-shop scheduling problem space (the Schedworld), for example, punching a hole through an object damages any polish on the object's surface. Thus, the goal of having a polished surface is clobbered by punching a hole through the object.

The notation $g|s$ denotes that the subgoal g holds in the state s, and $c(s)$ denotes the state that results from applying the operator sequence c to the state s. The notation $\Box C(g, pg)$ denotes that the goal g necessarily clobbers the protected goal pg.

$$\Box C(g, pg) \equiv \forall s \text{ s.t. } pg|s, \text{ and } \forall c \text{ s.t. } g|c(s), \text{ we have that } \neg pg|c(s).$$

Let $E(g, pg)$ denote the necessary effects of achieving the subgoal g from any state in which the protected subgoal pg holds. The necessary effects are the set of literals that necessarily hold in the current state after g is achieved by an arbitrary plan from any state in which pg holds.

$$E(g, pg) = \{e | \forall s \text{ s.t. } pg|s, \text{ and } \forall c \text{ s.t. } g|c(s), \text{ we have that } e|c(s)\}$$

$E(g, pg)$ is precisely the set of interest when considering whether one goal will clobber another. g clobbers pg if and only if pg is negated as a necessary effect of achieving g.

Lemma 1 $\Box C(g, pg)$ *iff* $\exists e \in E(g, pg)$ *s.t.* $e \rightarrow \neg pg$.[3]

STATIC anticipates goal interactions by computing a subset of of $E(g, pg)$ denoted by $\hat{E}(g)$. The necessary effects of achieving a goal are the necessary effects shared by all the operators that could potentially achieved the goal. The necessary effects (computed by STATIC) of an operator o that achieves g are denoted by $\hat{E}_o(g)$.

$$\hat{E}(g) = \bigcap_{o \in ops(g)} \hat{E}_o(g).$$

Thus, a goal is always one of its necessary effects (i.e. $\forall g$, we have that $g \in \hat{E}(g)$). The necessary effects of an operator are, in essence, the union of its effects and the necessary effects of its preconditions. When two effects are potentially contradictory only the one that would occur later during problem solving is retained.

$$\hat{E}_o(g) = \left[\bigcup_{p \in precs(o)} \hat{E}(p) \right] \cup effects(o).$$

STATIC's computation is correct. That is,

Theorem 1 $\hat{E}(g) \subseteq E(g, pg)$.

To illustrate the algorithm's operation consider the Schedworld goal `(shape Obj cylindrical)`. Only two operators can potentially achieve this goal: LATHE and ROLL. Both operators delete the surface condition of the object they are applied to. As a result, achieving `(shape Obj cylindrical)` necessarily clobbers `(surface Obj polished)`. STATIC detects this necessary side-effect by intersecting $\hat{E}_{lathe}$ and $\hat{E}_{roll}$. The rule formed based on the analysis appears in Table 3.

```
(PREFER-SHAPE
 (if (and
   (current-node N)
   (candidate-goal N (shape V2 cylindrical)))
   (candidate-goal N (surface V2 polished)))
 (then (prefer goal (shape V2 cylindrical)
                    (surface V2 polished)))
```

Table 3: A goal ordering rule produced by STATIC.

Comparing STATIC with PRODIGY/EBL

This section compares the impact, time required to generate control knowledge, and scope of STATIC and PRODIGY/EBL.[4] Although STATIC outperforms PRODIGY/EBL in the problem spaces studied, this will not always be the case. PRODIGY/EBL can exploit the distribution of problems encountered by PRODIGY whereas STATIC cannot and, unlike STATIC, PRODIGY/EBL can be guided by a carefully chosen training-example sequence.

Experimental Methodology

In his thesis [1988], Minton tested PRODIGY/EBL on one hundred randomly generated problems in three problem spaces (the Blocksworld, an extended version of the STRIPS problem space [Fikes *et al.*, 1972], and a machine-shop scheduling problem space) and showed that PRODIGY/EBL is able to significantly speed up PRODIGY. These problem spaces, as well as a modified version of the Blocksworld (the ABworld) constructed to foil PRODIGY/EBL, are used to compare STATIC and PRODIGY/EBL. PRODIGY is run on Minton's test-problem sets under three experimental conditions: guided by no control rules, guided by PRODIGY/EBL's rules, and guided by STATIC's rules. A bound of 150 CPU seconds was imposed on the solution time for each problem. The times reported below are slightly smaller than the times in [Etzioni, 1990b] due to minor improvements to the systems involved.

Impact

As Table 4 demonstrates, STATIC is able to speed up PRODIGY more than PRODIGY/EBL in each of Minton's problem spaces and in the ABworld. This section reports on problem-solving time, number of nodes expanded, and average time to expand a node. No significant differences were found in the length of the solutions produced by the systems. Since neither system attempts to reduce solution length, this is not surprising.

	PRODIGY	EBL	STATIC	Ratio
Blocksworld	2182	139	47	2.96
Stripsworld	4347	292	226	1.29
Schedworld	4391	1262	685	1.84
ABworld	869	925	711	1.30

Table 4: Total problem-solving time in CPU seconds. In the Schedworld, STATIC was unable to solve one problem within the 150-CPU second time bound compared with eight problems for PRODIGY/EBL and twenty three for PRODIGY.

The total number of nodes expanded by PRODIGY/EBL and STATIC is fairly close, compared with the number of nodes expanded by PRODIGY given

[3] Proofs of the results are in [Etzioni, 1990a].

[4] It is important to note that PRODIGY/EBL predates STATIC by several years and that STATIC is based on an in-depth study of PRODIGY/EBL.

no control rules (Table 5). The number of nodes expanded indicates the ability of each system to curtail PRODIGY's search. As the table shows, STATIC actually expanded more nodes than PRODIGY/EBL in the Blocksworld and the Stripsworld. Since STATIC's rules are cheaper-to-match than PRODIGY/EBL's (Table 6), however, STATIC's overall search time was consistently lower than PRODIGY/EBL's. STATIC's rules are relatively cheap-to-match for two reasons. First, STATIC does not learn from recursive explanations which tend to yield control rules that are both more specific and more expensive-to-match than their nonrecursive counterparts. Second, PRODIGY/EBL's control rules often retain incidental features of its training examples that do not appear in STATIC's rules. Although EBL only retains the aspects of its training example that appear in the weakest preconditions of its explanation, often there is a large set of possible explanations associated with any example, explanations whose weakest preconditions differ greatly. Frequently, STATIC's "explanations" are more compact than those produced by PRODIGY/EBL even when both are nonrecursive.

	PRODIGY	EBL	STATIC
Blocksworld	217,948	1689	1794
Stripsworld	219,349	4281	4911
Schedworld	181,938	5401	1654
ABworld	48,667	36,204	33,336

Table 5: Total number of nodes expanded.

	PRODIGY	EBL	STATIC
Blocksworld	0.010	0.082	0.026
Stripsworld	0.012	0.068	0.046
Schedworld	0.024	0.234	0.414
ABworld	0.018	0.025	0.021

Table 6: Average CPU seconds per node. The center and right columns reflect the cost of matching PRODIGY/EBL and STATIC's rule sets respectively.

Cost of Learning

The time required for learning is an important aspect of a learning method because, when learning time scales badly with problem size, learning can become prohibitively expensive on problems of interest. In the problem spaces studied, STATIC was able to generate control knowledge twenty six to four hundred and sixty three times faster than PRODIGY/EBL. Table 7 compares the learning time for the two systems. PRODIGY/EBL was "trained" by following the learning procedure outlined in Minton's thesis.

PRODIGY/EBL's learning time includes the time required to solve the training examples whose problem-solving traces are analyzed by PRODIGY/EBL. Table 8 decomposes PRODIGY/EBL's running time into three components: time to solve training examples, time to construct proofs based on the training examples' traces, and time to perform utility evaluation. It's interesting to note that utility evaluation is quite costly, and that STATIC's total running time is consistently smaller than PRODIGY/EBL's proof construction time.

	PRODIGY/EBL	STATIC	Ratio
Blocksworld	762	9.9	77.0
Stripsworld	1057	40.5	26.1
Schedworld	1374	19.4	70.8
ABworld	11,298	24.4	463.0

Table 7: Learning time in CPU seconds.

	Training examples	Proofs	Utility
Blocksworld	127	313	322
Stripsworld	305	671	81
Schedworld	454	832	88
ABworld	2493	8067	673

Table 8: PRODIGY/EBL's learning time decomposed into components: time to solve training examples, time to construct proofs based on the problem-solving trace, and time to perform utility evaluation.

Why did STATIC run so much faster than PRODIGY/EBL? STATIC traverses the PSGs for the problem space, whereas PRODIGY/EBL traverses PRODIGY's problem solving trace for each of the training examples. Since STATIC's processing consists of several traversals over its PSGs (construction, labeling, analysis of necessary effects, etc.), its running time is close to linear in the number of nodes in its PSGs. Since PRODIGY/EBL analyzes PRODIGY's traces, it learning time scales with the number of nodes in PRODIGY's traces. We can predict, therefore, that the number of trace nodes visited by PRODIGY/EBL is much larger than the number of nodes in STATIC's PSGs. Table 9 confirms this prediction. In fact, the ratio of trace nodes to PSG nodes is almost perfectly correlated (correlation=0.999) with the ratio of PRODIGY/EBL's running time to STATIC's.

	Trace nodes	PSG nodes	Ratio
Blocksworld	4834	86	56.2
Stripsworld	2360	186	12.7
Schedworld	4730	87	54.4
ABworld	143,954	249	578.1

Table 9: PSG nodes versus trace nodes.

It is reasonable to believe, therefore, that STATIC will continue to be significantly faster than PRODIGY/EBL when the ratio between trace size, summed over PRODIGY/EBL's training examples, and PSG size re-

mains large. PSGs are more compact than traces because the PSGs are partially instantiated, and because they do not contain nodes corresponding to recursive expansions.

Scope

STATIC utilizes target concepts based on those in PRODIGY/EBL, but only learns from nonrecursive explanations. Thus, STATIC will not acquire control knowledge that can only be learned from recursive explanations. Fortunately for STATIC, the knowledge required to control search in the problem spaces studied can usually be derived from nonrecursive explanations using STATIC's target concepts. The target concepts can frequently explain PRODIGY's behavior in multiple ways, some of which are recursive and some of which are not. PRODIGY/EBL acquires control rules from each of these explanations causing it to generate redundant control knowledge. STATIC's more conservative policy of only learning from nonrecursive explanations is advantageous in this case; STATIC learns fewer redundant rules than PRODIGY/EBL.

Because it performs static analysis, the range of proofs utilized by STATIC is narrower than that of PRODIGY/EBL. STATIC does not analyze state cycles for example, because, in contrast to PRODIGY's trace, no unique world state is associated with the PSG's nodes. Thus, merely *detecting* potential state cycles would be more difficult for STATIC than for PRODIGY/EBL.

STATIC only analyzes binary goal interactions as opposed to N-ary ones. The decision to analyze binary goal interactions is analogous to STATIC's policy of analyzing only nonrecursive explanations. In both cases additional coverage can be obtained by analyzing a broader class of explanations, but STATIC chooses to focus on the narrower class to curtail the costs of acquiring and utilizing control knowledge. In the problem spaces studied, STATIC acquires more effective control knowledge than PRODIGY/EBL despite (or, perhaps, due to) STATIC's narrower scope.

Polynomial-Time Problem Solving

By definition, polynomial-time problem solving can only be achieved for problem spaces that are in the complexity class P. No learning method will work for all problems in P, because finding a polynomial-time algorithm for an arbitrary problem in P is undecidable. A distribution-free sufficient condition for the success of STATIC (and EBL) can be derived, however, based on the notion of a *fortuitous* recursion. A recursion is said to be fortuitous when any plan that contains the recursion will succeed if the nonrecursive portion of the plan succeeds. When all problem-space recursions are fortuitous, nonrecursive proofs of failure suffice to eliminate backtracking, and STATIC can achieve polynomial-time problem solving.

Three additional definitions are required to state this idea precisely. Let a Datalog problem solver be a problem solver whose operator preconditions and effects are restricted to Datalog literals.[5] A goal interaction between two subgoals is said to be *binary* when it occurs independent of other subgoals. A goal interaction is said to be *nonrecursive* if it can be detected using a nonrecursive proof.

Theorem 2 STATIC *will achieve polynomial-time problem solving for a Datalog problem solver when:*

1. *All recursions are fortuitous.*
2. *All goal interactions are binary and nonrecursive.*
3. *Solution length is polynomial in state size.*

When the above conditions hold in a given problem space, STATIC's exhaustive generation of "nonrecursive control knowledge" guarantees that it will eliminate backtracking. Since, by assumption, solution length is polynomial in state size, it follows that no more than a polynomial number of search-tree nodes will be expanded. Since the cost of matching STATIC's control knowledge at each node is polynomial in the state size (see [Etzioni, 1990a, Chapter 3]), STATIC will achieve polynomial-time problem solving. EBL will achieve polynomial-time problem solving when it has the appropriate target concepts (failure and goal interaction) and it encounters the appropriate training examples; namely, the ones that enable EBL to eliminate backtracking using nonrecursive proofs.

Related Work

Although STATIC utilizes partial evaluation to construct its PSGs, STATIC goes beyond standard partial evaluators in several important ways. First, STATIC analyzes failure and goal interaction whereas standard partial evaluators do not. Second, STATIC derives various subgoal reordering rules from this analysis. Such reordering transformations are not part of the usual arsenal of partial evaluators. Third, STATIC introduces a novel criterion for choosing between multiple target concepts: *In the absence of knowledge about problem distribution, learn only from the target concepts that yield nonrecursive proofs.* This criterion addresses the problem of recursive explanations for STATIC, a topic of active research for the PE community [van Harmelen, 1989, Sestfot, 1988].

Unlike STATIC, most partial evaluators do not analyze goal interactions. The remainder of this section compares STATIC with other problem space analyzers that do attempt to anticipate and avoid goal interactions.

[5] Datalog is the function and negation free subset of pure Prolog [Ullman, 1989]. A Datalog problem solver represents a slight idealization of the PRODIGY problem solver.

REFLECT Dawson and Siklossy [1977] describe REFLECT, an early system that engaged in static problem space analysis. REFLECT ran in two phases: a preprocessing phase in which conjunctive goals were analyzed and macro-operators were built followed by a problem-solving phase in which REFLECT engaged in a form of backward-chaining. During its preprocessing phase REFLECT analyzed operators to determine which pairs of achievable literals (e.g., `(holding X)`, `(arm-empty)`) were incompatible. This was determined by symbolically backward-chaining on the operators up to a given depth bound. A graph representation, referred to as a goal-kernel graph, was constructed in the process. The nodes of the goal-kernel graph represent sets of states, and its edges represent operators. REFLECT also constructed macro-operators using heuristics such as "requiring that at least one variable in both operators correspond at all times."

Although the actual algorithms employed by REFLECT and STATIC are different, the spirit of STATIC is foreshadowed by this early system. In particular, STATIC's PSG is reminiscent of the goal-kernel graph, both systems analyze successful paths, and both system analyze pairs of achievable literals. One of the main advances found in STATIC is its theoretically motivated and well-defined criteria for terminating symbolic back-chaining, particularly its avoidance of recursion. Another advance is the formal specification and computation of necessary effects and prerequisites to facilitate goal reordering. Finally, unlike STATIC, REFLECT does not actually reorder goals. REFLECT merely terminates backward-chaining when an incompatible set of goals is reached.

ALPINE Knoblock's ALPINE [1991] generates hierarchies of abstract problem spaces by statically analyzing goal interactions. ALPINE maps the problem space definition to a graph representation in which a node is a literal, and an edge from one literal to another means that the first literal can potentially interact with the former. Knoblock's guarantees on ALPINE's behavior are analogous to my own: I compute a *subset* of the *necessary* goal interactions, and Knoblock computes a *superset* of the *possible* goal interactions. Both algorithms are correct and conservative. The ABSOLVER system [Mostow and Prieditis, 1989] contains a transformation that computes a superset of possible goal interactions as well.

Universal Plans Schoppers [1989] derives "universal plans" from problem space descriptions. Universal plans are, essentially, sets of rules that tell an agent what to do in any given state. Schoppers partially evaluates the problem space definition to derive universal plans. Like STATIC, Schoppers only considers pairwise goal interactions. Unlike STATIC, he ignores the issues of recursion and binding-analysis for partially instantiated goals by only considering tightly circumscribed problem spaces such as the 3-block Blocksworld where recursion is bounded and variables are fully instantiated. Instead, Schoppers focuses on handling conditional effects, nonlinear planning, and incorrect world models.

Problem-Specific Methods Whereas STATIC analyzes the problem space definition, some algorithms analyze individual problems. Whereas STATIC is only run once per problem space, the algorithms described below are run anew for each problem. Since the algorithms use problem-specific information, they are often able to make strong complexity and completeness guarantees presenting a tradeoff between problem-space analysis and problem-specific analysis. This tradeoff has not been studied systematically.

Chapman's TWEAK [1987] utilizes an algorithm for determining the necessary/possible effects of a partially specified plan. The input to the algorithm is an initial state and the plan. The output of the algorithm is a list of the plan's necessary (occurring in *all* completions of the plan) and possible (occurring in *some* completion of the plan) effects. The algorithm runs in time that is polynomial in the number of steps in the plan. Cheng and Irani [1989] describe an algorithm that, given a goal conjunction, computes a partial order on the subgoals in the goal conjunction. If a problem solver attacks the problem's subgoals according to the specified order, it will not have to backtrack across subgoals. The algorithm runs in $O(n^3)$ time where n is the number of subgoals. Finally, Knoblock's ALPINE performs some initial preprocessing of the problem space definition, but then develops considerably stronger constraints on possible goal interactions by analyzing individual problems.

Conclusion

This paper addresses the questions posed at the outset by means of a case study. Partial evaluation à-la-STATIC can be performed in reasonable time on PRODIGY/EBL's domain theories. The problem of recursion is overcome by learning only from nonrecursive explanations utilizing multiple target concepts. The control knowledge derived by STATIC is better than that acquired by PRODIGY/EBL, and was generated considerably faster. Finally, STATIC's analysis of goal interactions demonstrates that partial evaluation can be applied to planning as well as inference tasks.

As the previous section indicates, STATIC is only one point in the space of static problem space analyzers. Consider Knoblock's ALPINE [1991] as a point of contrast. STATIC is run once per problem space whereas ALPINE preprocesses individual problems. STATIC analyzes *necessary* goal interactions whereas ALPINE analyzes *possible* goal interactions. STATIC outputs control rules whereas ALPINE outputs abstraction hierarchies. The differences between the two systems suggest that the space of static analyzers is large and diverse. The

demonstrated success of the two systems indicates the potential of static problem space analysis as a tool for deriving control knowledge. Ultimately, this success is not surprising. Powerful optimizing compilers have been developed for most widely-used programming languages. There is no reason to believe that problem-space-specification languages will be an exception.

Acknowledgments

This paper describes research done primarily at Carnegie Mellon University's School of Computer Science. The author was supported by an AT&T Bell Labs Ph.D. Scholarship. Thanks go to the author's advisor, Tom Mitchell, and the members of his thesis committee, Jaime Carbonell, Paul Rosenbloom, and Kurt VanLehn for their contributions to the ideas herein. Special thanks go to Steve Minton—whose thesis work made studying PRODIGY/EBL possible, and to Ruth Etzioni for her insightful comments.

References

Carbonell, J. G. and Gil, Y. 1990. Learning by experimentation: The operator refinement method. In *Machine Learning, An Artificial Intelligence Approach, Volume III.* Morgan Kaufman, San Mateo, CA. 191–213.

Chapman, David 1987. Planning for conjunctive goals. *Artificial Intelligence* 32(3):333–378.

Cheng, Jie and Irani, Keki B. 1989. Ordering problem subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan. Morgan Kaufmann. 931–936.

Dawson, Clive and Siklossy, Laurent 1977. The role of preprocessing in problem-solving systems. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

Dejong, G. F. and Mooney, R. J. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(1).

Etzioni, Oren 1990a. *A Structural Theory of Explanation-Based Learning.* Ph.D. Dissertation, Carnegie Mellon University. Available as technical report CMU-CS-90-185.

Etzioni, Oren 1990b. Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence.*

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4).

Joseph, Robert L. 1989. Graphical knowledge acquisition. In *Proceedings of the Fourth Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada.

Knoblock, Craig; Minton, Steve; and Etzioni, Oren 1991. Integrating abstraction and explanation-based learning in prodigy. In *Proceedings of the Ninth National Conference on Artificial Intelligence.*

Knoblock, Craig A. 1991. *Automatically Generating Abstractions for Problem Solving.* Ph.D. Dissertation, Carnegie Mellon University.

Minton, Steven; Carbonell, Jaime G.; Knoblock, Craig A.; Kuokka, Daniel R.; Etzioni, Oren; and Gil, Yolanda 1989a. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40:63–118. Available as techincal report CMU-CS-89-103.

Minton, Steven; Knoblock, Craig A.; Kuokka, Daniel R.; Gil, Yolanda; Joseph, Robert L.; and Carbonell, Jaime G. 1989b. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Carnegie Mellon University.

Minton, Steven 1988. *Learning Effective Search Control Knolwedge: An Explanation-Based Approach.* Ph.D. Dissertation, Carnegie Mellon University. Available as technical report CMU-CS-88-133.

Mitchell, Tom M.; Keller, Rich; and Kedar-Cabelli, Smadar 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1(1).

Mostow, Jack and Prieditis, Armand E. 1989. Discovering admissible heuristics by abstracting and optimizing: a transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.*

Prieditis, A. E. 1988. Environment-guided program transformation. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.*

Schoppers, Marcel Joachim 1989. *Representation and Automatic Synthesis of Reaction Plans.* Ph.D. Dissertation, University of Illinois at Urbana-Champaign. Available as technical report UIUCDCS-R-89-1546.

Sestfot, Peter 1988. Automatic call unfolding in a partial evaluator. In Bjorner, D.; Ershov, A. P.; and Jones, N. D., editors 1988, *Partial Evaluation and Mixed Computation.* Elsevier Science Publishers. Workshop Proceedings.

Ullman, Jeffrey D. 1989. *Database and Knowledge-base systems*, volume I. computer science press.

van Harmelen, Frank and Bundy, Alan 1988. Explanation-based generalisation = partial evaluation. *Artificial Intelligence* 36. Research Note.

van Harmelen, Frank 1989. The limitations of partial evaluation. In *Logic-Based Knowledge Representation.* MIT Press, Cambridge, MA. 87–112.

Veloso, Manuela M. and Carbonell, Jaime G. 1990. Integrating analogy into a general problem-solving architecture. In *Intelligent Systems.* Ellis Horwood Limited.