Acquiring Search-Control Knowledge via Static Analysis*

Oren Etzioni

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

Explanation-Based Learning (EBL) is a widely-used technique for acquiring search-control knowledge. Recently, Prieditis, van Harmelen, and Bundy pointed to the similarity between Partial Evaluation (PE) and EBL. However, EBL utilizes training examples whereas PE does not. It is natural to inquire, therefore, whether PE can be used to acquire search-control knowledge, and if so at what cost? This paper answers these questions by means of a case study comparing PRODIGY/EBL, a state-of-the-art EBL system, and STATIC, a PE-based analyzer of problem-space definitions. When tested in PRODIGY/EBL's benchmark problem spaces, STATIC generated search-control knowledge that was up to three times as effective as the knowledge learned by PRODIGY/EBL, and did so from twenty-six to seventy-seven times faster. The paper describes STATIC's algorithms, compares its performance to PRODIGY/EBL's, noting when STATIC's superior performance will scale up and when it will not. The paper concludes with several lessons for the design of EBL systems, suggesting hybrid PE/EBL systems as a promising direction for future research.

*STATIC is available by sending mail to the author at `etzioni@cs.washington.edu`. The PRODIGY system, and the information necessary to replicate the experiments in this paper, is available by sending mail to `prodigy@cs.cmu.edu`.

1 Introduction

Explanation-Based Learning (EBL) [11, 36] has emerged as a standard technique for acquiring search-control knowledge (e.g. [27, 33, 37, 35, 47, 52]).[1] Recently, Prieditis [46], van Harmelen and Bundy [59] pointed to the similarity between Partial Evaluation (PE), a well-known program optimization technique [25, 58], and EBL, suggesting that an EBL-style analysis could be performed *statically*, without utilizing training examples. This suggestion is controversial because EBL is perceived as a dynamic technique that depends on examples to focus the learning process. It is natural to inquire, therefore, whether search-control knowledge can be acquired via PE-based static analysis.

Static analysis has three advantages over standard EBL methods. First, while EBL is sensitive to incomplete or badly-ordered sequences of training examples, static analysis is not. Second, EBL often acquires overly-specific control knowledge by retaining extraneous features of its training examples [16, 44]. Finally, in contrast to EBL, a static analyzer need not invoke the problem solver to solve training problems—a costly process. However, because training problems "transmit" useful information about the task environment, it is not obvious that static analysis can actually generate effective control knowledge and do so tractably. Thus, we arrive at two fundamental questions:

- Can static analysis produce effective search-control knowledge?
- Can it do so in reasonable time?

This paper presents a case study, using the PRODIGY system [33], which answers these questions. Since a state-of-the-art EBL system, PRODIGY/EBL, has already been implemented and tested, our task has been to design a PE-based static analyzer and compare its performance with that of PRODIGY/EBL. PRODIGY/EBL was tested on sets of one-hundred randomly generated problems in three benchmark problem spaces [31]. The performance of the static analyzer (called STATIC) was compared to that of PRODIGY/EBL, using the same sets of test problems, in each of the benchmark problem spaces. Surprisingly, STATIC generated search-control knowledge that was up to three times as effective as PRODIGY/EBL's, and did so from twenty-six to seventy-seven times faster. The paper explains these experimental results, and considers under what conditions the results will be replicated in larger and more complex problem spaces.

The paper is organized as follows. Section 2 provides necessary background on EBL, PRODIGY, and PE. The subsequent sections describe STATIC's algorithms in detail, and compare it to PRODIGY/EBL. Section 6 contrasts STATIC with standard partial evaluators and other static analyzers, and Section 7 concludes by considering design lessons for EBL systems based on the STATIC case study.

[1] EBL is also used to revise or repair "imperfect" theories (e.g. [8, 20, 40, 42, 48]).

2 Background

This section is organized as follows. Section 2.1 describes the PRODIGY problem solver, which is the substrate for PRODIGY/EBL. Section 2.2 introduces standard EBL terminology, and sketches PRODIGY/EBL. Finally, Section 2.3 defines partial evaluation.

2.1 The PRODIGY Problem Solver

Detailed descriptions of PRODIGY appear in [29, 33, 34]. The bare essentials follow. PRODIGY is a domain-independent problem solver. Given an initial state and a goal expression, PRODIGY searches for a sequence of operators that will transform the initial state into a state that matches the goal expression. A sample PRODIGY operator appears in Table 1. PRODIGY's sole problem-solving method is a form of means-ends analysis [39]. Like STRIPS [17], PRODIGY employs operator preconditions as its differences. However, PRODIGY's operator description language is considerably more expressive, allowing universal quantification and conditional effects.

```
(UNSTACK
    (preconditions
        (and (object Block-X) (object Block-Y)
             (on Block-X Block-Y) (clear Block-X) (arm-empty)))
    (effects ((del (on Block-X Block-Y))
              (del (clear Block-X))
              (del (arm-empty))
              (add (holding Block-X))
              (add (clear Block-Y)))))
```

Table 1: The Blocksworld operator UNSTACK. Variable names are capitalized.

PRODIGY's default search strategy is depth-first search. The search is carried out by repeating the following decision cycle [30]:

1. Choose a node in the search tree. A node consists of a set of goals and a world state.
2. Choose one of the goals at that node.
3. Choose an operator that can potentially achieve the goal.
4. Choose bindings for the variables in the operator. If the instantiated operator's preconditions match the state then apply the operator and update the state, otherwise subgoal on the operator's unmatched preconditions. In either case, a new node is created.

Search-control knowledge in PRODIGY is encoded via control rules, which override PRODIGY's default behavior by specifying that particular candidates (nodes, goals, operators, or bindings) should be selected, rejected, or preferred over other candidates [30]. Alternatives that are selected are the only ones tried; alternatives that are rejected are removed from the selected set. Finally, all other things being equal, preferred alternatives are tried before other ones. PRODIGY matches control rules against its current state. If the antecedent of a control rule matches, PRODIGY abides by the recommendation in the consequent. For example, the control rule in Table 2 tells PRODIGY to reject the Blocksworld operator UNSTACK when the block to be held is not on any other block.

```
(REJECT-UNSTACK
  (if (and (current-node Node)
           (current-goal Node (holding Block-X))
           (candidate-operator Node unstack)
           (known Node (not (on Block-X Block-Y)))))
  (then (reject operator unstack)))
```

Table 2: A control rule from PRODIGY's Blocksworld.

2.2 Explanation-Based Learning

Mitchell, Keller and Kedar-Cabelli describe a model of EBL, called Explanation-Based Generalization (EBG) [36], which articulates many of the aspects common to various EBL systems. Two of the major contributions of the EBG model are the identification of explanations with proofs, giving a precise meaning to the term "explanation," and the clear specification of the input and output of EBL (Table 3). The input consists of a target concept, a theory for constructing explanations, a training example, and an operationality criterion. The output of EBL is a sufficient condition for recognizing the target concept. The operationality criterion is intended to ensure that the sufficient condition can be used to recognize instances of the concept efficiently. EBL proves (or explains) that the training example is an instance of the target concept and outputs the weakest preconditions [12] of the proof.

2.2.1 PRODIGY/EBL

A terse description of PRODIGY/EBL, using EBG terminology, follows. A complete description of PRODIGY/EBL appears in [29]. PRODIGY/EBL's primary *target concepts* are success, failure and goal interaction. PRODIGY/EBL analyzes a trace of PRODIGY's search on a training problem, and extracts *training examples* from this trace. PRODIGY/EBL's *domain theory* has two components: the architectural theory, which describes PRODIGY's problem-solving

Given:

- *Target Concept Definition:* A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the Operationality Criterion.)
- *Training Example:* An example of the target concept.
- *Domain Theory:* A set of rules and facts to be used in explaining how the training example is an example of the target concept.
- *Operationality Criterion:* A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalization of the training example that is a sufficient concept description for the target concept and that satisfies the operationality criterion.

Table 3: Mitchell *et al.*'s specification of EBG.

method declaratively, and the problem-space definition which consists of PRODIGY's operators, and axioms constraining the set of legal problem-space states. PRODIGY/EBL's *operational* predicates are problem-space predicates (e.g. `on-table` in the Blocksworld), and a small set of predicates that describe the problem solver's state (e.g. `current-goal`). PRODIGY/EBL proves that various candidates (e.g. operators) will succeed, fail, or interact. Its output consists of control rules whose antecedents are the operational weakest preconditions of its proofs, and whose consequents expresses preferences based on the proofs' conclusions. For example, a proof that an operator will fail, results in a control rule that rejects that operator.

PRODIGY/EBL attempts to reduce the match cost of the rules it generates by a combination of logical simplification and theorem proving. This process is referred to as *compression.* PRODIGY/EBL also estimates the utility of learned rules (both empirically and analytically) and discards rules that are deemed to be harmful. This process is referred to as *utility evaluation.* In summary, PRODIGY/EBL extends traditional EBL methods in three important ways. First, PRODIGY/EBL utilizes multiple target concepts. Second, PRODIGY/EBL derives control rules based on the sufficient conditions it computes. Third, PRODIGY/EBL performs post-processing of the control rules in an effort to increase their effectiveness.

2.3 Partial Evaluation

Partial evaluation is a program optimization technique inspired by Kleene's S-M-N theorem [22, 59]. The theorem states that given any computable function f over n variables, $f(x_1, ..., x_n)$, k of which have known values $a_1, ..., a_k$, we can effectively compute a new func-

tion f' which only takes $n-k$ inputs such that:

$$f'(x_{k+1}, ..., x_n) = f(a_1, ..., a_k, x_{k+1}, ..., x_n)$$

Presumably, the new function f' is easier to compute than f.

Partial evaluation is typically implemented by symbolically executing the program that computes f and performing, at compile-time, any computation whose value depends only on the known inputs. The fundamental operation performed, when symbolically executing a program, is known as "call unfolding" or simply *unfolding*. Unfolding replaces a function call in the program by the function's body, substituting the values of the function's arguments for its variables. When the values of the arguments are unknown, place-holders or "symbolic values," are created and propagated through the program. In a language such as Prolog, unfolding replaces an atomic formula in the body of one Horn clause by the body of another clause whose head unifies with that formula. For example, given the following Horn clauses:

```
liftable(Z) :- has-handle(Z), small(Z).
     cup(X) :- liftable(X), liquid-container(X).
```

The formula `liftable(X)` could be unfolded to derive:

```
cup(X) :- has-handle(X), small(X), liquid-container(X).
```

We can view a problem solver such as PRODIGY as a function P that maps a problem-space definition d, an initial state i, and a goal expression e, to an operator sequence s that transforms the initial state into one which satisfies the goal expression. That is, $P(d, e, i) = s$. Partial evaluation, in this context, amounts to symbolically executing the program for P with some of its inputs unknown. In particular, STATIC takes the problem space definition d as input and symbolically back-chains on the uninstantiated operators in d. Thus, we can view PRODIGY operators as functions being unfolded.

A critical issue in designing a partial evaluator is finding an appropriate criterion for terminating the unfolding process. If the *termination criterion* is too conservative (e.g. never unfold calls) no optimization will be achieved, but if the criterion is too liberal (e.g. unfold all calls) the partial evaluator will fail to halt when unfolding recursive calls [51]. As van Harmelen and Bundy demonstrate, EBL algorithms unfold the domain theory guided by their training example and operationality criterion. This elaboration of PE is important because it provides EBL with a termination criterion. The theory is unfolded in accord with the choices made in the training example. A partial evaluator will only match EBL's performance if it is given an appropriate termination criterion. In [59], van Harmelen and Bundy describe a simple partial evaluator that is able to emulate EBL, but they do not specify an appropriate termination criterion.

STATIC solves this problem by combining (its own versions of) PRODIGY/EBL's multiple target concepts with a simple termination criterion: *unfold only nonrecursive calls.*[2] As shown in Section 3.2.2, STATIC provably terminates, yet it is still able to outperform PRODIGY/EBL (Section 5).

[2] Theoretical motivation for this criterion appears in [15].

3 Static Analysis of Failure and Success

This section describes STATIC's algorithms for analyzing failure and success. The algorithms in this section could be used to optimize pure Prolog programs. Section 4 presents STATIC's analysis of goal interactions. The algorithms in Section 4 are specific to problem solvers, like PRODIGY, that transform states "nonmonotonically," by adding and deleting atomic formulas.

STATIC's input is a problem-space definition, consisting of PRODIGY operators and axioms constraining legal states; its output is a set of PRODIGY control rules. Unlike PRODIGY/EBL, STATIC does not construct explanations or even utilize an explicit domain theory. Instead, STATIC symbolically back-chains on PRODIGY's operators to construct an AND/OR graph representation of the problem space, known as a Problem Space Graph (PSG). Essentially, the PSG represents all backward-chaining paths through the problem space. To keep the PSG finite, STATIC restricts its attention to nonrecursive paths, terminating PSG expansion whenever recursion is encountered. STATIC annotates each PSG node with a label, indicating which operators and subgoals will succeed or fail, and with a logical expression indicating under what conditions the label holds. STATIC derives PRODIGY control rules based on this annotation.

Section 3.1 defines the PSG representation precisely, followed by a description of PSG construction, labeling, and analysis.

3.1 Problem Space Graphs (PSGs)

A PSG is a directed, acyclic AND/OR graph that represents the goal/subgoal relationships in a problem space. Each PSG is rooted in a distinct goal literal.[3] The root literal is connected, via OR-links, to the operators that can potentially achieve it. An operator can potentially achieve a literal when one of its effects unifies with the literal. Each operator in the PSG is partially instantiated via the variable substitution from this unification. Each operator is connected, via AND-links, to its partially instantiated preconditions. Each precondition is connected to the operators that can potentially achieve it and so on. Thus, the PSG's nodes are an alternating sequence of subgoals and operators, and the PSG's edges are an alternating sequence of AND-links and OR-links. Figure 1 depicts the Blocksworld PSG rooted in the subgoal `(holding V)`.

When two sibling operators share a precondition, a single node designates that precondition in the PSG. Both operators are linked to the node via AND-links, so the graph is not a tree. In Figure 1, for example, both PICK-UP(V) and UNSTACK(V,V2) have `(clear V)` as a precondition. Consequently, both operators have AND-links to that node.

The PSG is independent of any state information, and should not be confused with a state space graph. Its nodes are *not* states and its edges are *not* fully-instantiated operators. If we think of operators as functions being unfolded, the PSG can be viewed as a "call graph" for PRODIGY [38]. Although the PSG was developed independently, it is quite

[3] A literal is a possibly negated atomic formula.

similar to the rule/goal graph [57], and related graph representations of logic programs (e.g., [26]). In contrast to previous work, however, STATIC utilizes PSGs to compute the weakest preconditions of EBL-style proofs.

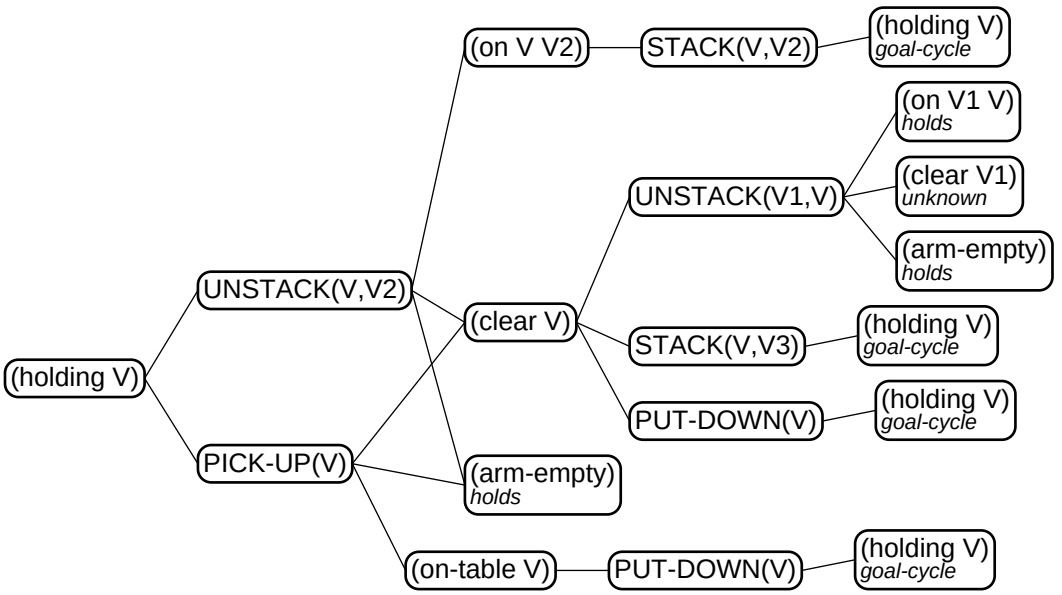

Figure 1: The `holding` PSG.

As described thus far, recursion would result in infinite PSGs. In fact, the PSG has well-defined termination conditions under which it is provably finite. To state these conditions precisely I introduce the notion of a literal's *ancestors*, which are the literals at the unique set of subgoal nodes on a path from the literal to the root of the PSG. The ancestors of a PSG node represent the set of subgoals PRODIGY would generate before subgoaling on the literal at the node. PSG expansion is terminated at a node if and only if:

- No operators can achieve the literal at the node.
- The literal is identical to one of its ancestor literals. I refer to this condition as a *goal cycle*.
- The literal unifies with, but is not identical to, one of its ancestors. I say that the literal recurs and label the node *unknown*.
- The literal is necessarily satisfied, given its ancestor subgoals. In the `holding` PSG, for example, `(on V1 V)` is labeled *holds* because, to reach it, PRODIGY subgoals on `(holding V)` and `(clear V)` and, if a block is neither clear nor held, then there must be some block on it. See Section 3.2.1 for more details.

Note that the above conditions apply to subgoals, not to operators. Thus, the second occurrence of UNSTACK in Figure 1 is *not* an instance of recursion or of a goal-cycle. Table 4 presents a simple algorithm that derives PSGs from problem space definitions.

3.2 Constructing PSGs

STATIC relies on partial evaluation to construct PSGs. Specifically, PSGs are constructed by symbolically back-chaining on the problem space's operators from the root literal. Each operator is partially (but not fully) instantiated since its effects are matched against a goal, but its preconditions are not matched against a state. The PSG construction algorithm creates a distinct PSG for each uninstantiated achievable literal in the problem space. STATIC finds the achievable literals by scanning the effects of the problem-space operators. A capsule description of the algorithm appears in Table 4.

Each operator node in the PSG contains the following information: the operator's name, an OR-link to its parent subgoal, AND-links to its partially instantiated preconditions, its label, and its partially instantiated effects. Each subgoal node in the PSG contains the following information: the partially instantiated literal, OR-links to the operators that achieve the literal, its ancestors, and its label.

3.2.1 Constraints on Legal States

STATIC's input is a problem-space definition, which consists of a set of operators and a set of constraints on legal states. The constraints rule out impossible states. In the Blocksworld, for example, either `(arm-empty)` is true or the robot is holding some block. STATIC utilizes the constraints on legal states to terminate the expansion of the PSG at certain subgoal nodes. The constraints are used to prove that, given the subgoal's ancestors, if the subgoal arises during actual problem solving it would invariably be matched by PRODIGY's current state. For example, suppose that the literal `(arm-empty)` is an ancestor of the subgoal `(holding V)`. We know that `(arm-empty)` is not true because PRODIGY is trying to achieve it. Therefore, the robot is holding *some* block and `(holding V)` will match the current state.

In general, we know that ancestor literals do not match PRODIGY's current state. Thus, the ancestor set provides information about the current state. That information, in conjunction with the constraints on legal states, is used by the function *holds(p,ancestors(p))* to determine whether p will be matched by PRODIGY's current state, whenever p arises in the current goal context. Constraints on legal states in the Blocksworld appear in Table 5. Exactly one of the literals in each constraint is true in every state. To use the constraints appropriately, STATIC reasons about the potential bindings for the variables at PSG nodes. Suppose, for example, that `(holding V)` is an ancestor of `(arm-empty)`. Since `V` may be bound to a constant b_1 during actual problem solving, the `holding` goal may not be satisfied, but `(arm-empty)` may still be false because the robot is holding yet another block b_2. Thus, we cannot conclude that `(arm-empty)` *holds*, and the PSG shown in Figure 1 is actually a simplified version of the "true" PSG. This simplification is inconsequential because `(arm-empty)` can always be achieved via PUT-DOWN. The only precondition to

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

The ancestors of a node n in the PSG, *ancestors(n)*, represent the set of subgoals PRODIGY would generate before subgoaling on the literal at the node. The function *holds(p,ancestors(p))*, which is described in more detail in the text, determines whether the precondition p necessarily holds given its ancestors and the constraints on legal states. Finally, $ops(g)$ refers to the operators that can potentially achieve g, and $precs(o)$ refers to the preconditions of the operator o.

Algorithm:

1. Create a subgoal node for g.
2. For each partially instantiated operator $o \in ops(g)$:
   (a) Create an operator node for o and OR-link it to g.
   (b) If any of o's preconditions is identical to one of o's ancestors, then create a subgoal node corresponding to that precondition, label it *goal-cycle*, and AND-link it to o.
   (c) Else, for each precondition $p \in precs(o)$:
      i. If a previously expanded sibling operator shares p, then a node for p already exists. AND-link o to the existing node.
      ii. Otherwise, create a new node p and AND-link it to o.
      iii. If $holds(p, (\text{ancestors}(p))$ then label p *holds*.
      iv. If p unifies with, but is not identical to, one of its ancestors, 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 4: Constructing a PSG.

PUT-DOWN is that the robot be holding *some* block and this *is* guaranteed to be the case when `(arm-empty)` is an ancestor goal.

$\forall X, \exists Y$ such that $\Omega(\text{on-table}(X), \text{on}(X, Y), \text{holding}(X))$.
$\exists X$ such that $\Omega(\text{arm-empty}, \text{holding}(X))$.
$\forall X, \exists Y$ such that $\Omega(\text{holding}(X), \text{clear}(X), \text{on}(Y, X))$.

Table 5: Constraints on Blocksworld states. Ω denotes a generalized exclusive-or; exactly one of Ω's arguments is true in every state.

Binding analysis of this sort is a widely-used technique in program optimization [57, chapter 12]. In fact, correctly terminating the expansion of PSGs is an instance of a general problem in the symbolic execution of programs known as "detecting spurious execution paths." Spurious execution paths are symbolically executed computations that would never occur during *actual* program execution because they are logically inconsistent with the semantics of the program. Expanding a subgoal node in the PSG, which PRODIGY would never actually subgoal on, is an example of a spurious execution path, Avoiding spurious paths is undecidable, in general, because detecting such paths can involve arbitrary inference about the program. The *holds* function is a simple heuristic that prunes some of the spurious execution paths in STATIC's PSGs. When this heuristic fails, STATIC will analyze spurious execution paths, and learn inapplicable control knowledge. However, even when this occurs, STATIC will not generate control rules that lead PRODIGY to overlook possible solutions to its problems.

3.2.2 PSG Size

To be a viable representation, a PSG must be compact. This section relates PSG size to problem space characteristics and shows that, in the problem spaces studied, PSGs are very compact indeed.

Several observations about PSGs follow from their definition in Section 3.1:

- The OR-link branching factor is bounded by the maximal number of operators that can potentially achieve any given literal.
- The AND-link branching factor is bounded by the maximal number of preconditions to any given operator.
- Since only one predicate repetition is allowed before the PSG's expansion is terminated, the depth of the PSG is bounded by $2p + 1$, where p is the number of predicates in the problem space definition. This bound is tight only when every predicate in the problem space participates in a given nonrecursive subgoaling chain. In practice, PSG depth tends to be much smaller.

In general, we have the following:

Proposition 1 *The PSG representation of any finite problem-space definition is finite.*

Note that because STATIC's algorithm amounts to repeated traversals of the PSG, Proposition 1 implies that STATIC is guaranteed to terminate. In fact, in the problem spaces studied, the PSGs are very compact (Table 6) which explains STATIC's speed.

Blocksworld	86
Stripsworld	186
Schedworld	87
ABworld	249

Table 6: Total number of PSG nodes in the problem spaces studied, which include PRODIGY/EBL's benchmark problem spaces and the ABworld, a highly-recursive variant of the Blocksworld constructed to foil PRODIGY/EBL [13, 14].

3.3 Labeling the PSG

Each PSG node, below the root, is annotated with a label and a logical expression referred to as its "failure condition." The label asserts what PRODIGY's behavior will be when an instance of the node is encountered during problem solving. For example, the label *holds* means that the literal at the node will be satisfied in the current state, and the label *goal-cycle* means that PRODIGY will find itself in a goal-cycle and backtrack. STATIC asserts that PRODIGY will fail at an instance of a PSG node only when the PSG node is labeled *failure* and the failure condition at the PSG node is true. For instance, an operator node labeled *failure* yields a control rule that recommends rejecting the operator when the node's failure condition is matched by PRODIGY's current state (see Table 9 for illustrations).

This section describes how STATIC computes the labels and associated failure conditions for a PSG. The following section explains how this information is used to derive control rules. The PSG labeling algorithm is described in two parts. First, the computation of the labels is explained, followed by the derivation of the failure conditions.

Each PSG leaf is labeled by one of the following labels when the PSG is constructed: *holds, unknown, goal-cycle*, or *unachievable*. Each internal PSG node is labeled *failure, success*, or *unknown*. The PSG is traversed in postorder to determine the labels of internal nodes. The label of each node is derived from the labels of its children using a three-valued "label logic" where *failure* corresponds to false, *success* corresponds to true, and *unknown* to unknown. The label *holds* is considered to be a *success* label, and the labels *goal-cycle* and *unachievable* are *failure* labels. The label of an operator is determined by the conjunction of the labels of its preconditions; the label of a precondition is determined by the disjunction of the labels of the operators that can potentially achieve it. The precise meaning of conjunction

and disjunction in label logic is defined by Table 7. When a set of nodes shares the same label, applying disjunction and conjunction to the set yields that label as well. If both L_1 and L_2 are $failure$, for example, then their conjunction and disjunction is $failure$ as well.

L_1	L_2	disjunction(L_1, L_2)	conjunction(L_1, L_2)
$failure$	$success$	$success$	$failure$
$failure$	$unknown$	$unknown$	$failure$
$unknown$	$success$	$success$	$unknown$

Table 7: Label logic.

If we think of false, unknown, and true as ordered, where false is minimal and true is maximal, then labeling the PSG amounts to a min-max search of the graph. Minimizing corresponds to conjunction, and maximizing corresponds to disjunction. In fact, an analog of $\alpha\beta$ pruning is possible. For example, once an operator is labeled *success*, the labels of sibling operators need not be computed to determine the label of the goal. $\alpha\beta$ pruning was not implemented in STATIC because, as Section 5 documents, constructing and labeling PSGs is already very fast in the problem spaces studied.

3.3.1 Deriving Failure Conditions

$$fprecs(o) = \{p \in precs(o) | label(p) = failure\}$$

$$\text{op-fc}(o, g, G) = \text{If } label(o) \neq failure \text{ then return } false.$$
$$\text{Else if } g \in G \text{ then return } true.$$
$$\text{Else return } \bigvee_{p \in fprecs(o)} \text{goal-fc}(p, G + g)$$

$$\text{goal-fc}(p, G) = \neg p \wedge \left(\bigwedge_{o \in ops(p)} \text{op-fc}(o, p, G) \right)$$

Table 8: STATIC's algorithm for computing failure conditions.

The failure conditions are logical expressions associated with PSG nodes labeled *failure*. The conditions are incrementally constructed as the PSG is labeled. Like a label, the failure condition of each PSG node is derived from the failure conditions of its children. The failure condition of an operator node is the disjunction of the failure conditions of its preconditions. The failure condition of a precondition node is the conjunction of the failure conditions of its

operators. Table 8 outlines the recursive procedure used to compute PSG failure conditions. A pseudo-code description of the algorithm and a trace of its execution appear in [13, chapter 6]. The output of the algorithm applied to the `holding` PSG depicted in Figure 1, appears in Figure 2.

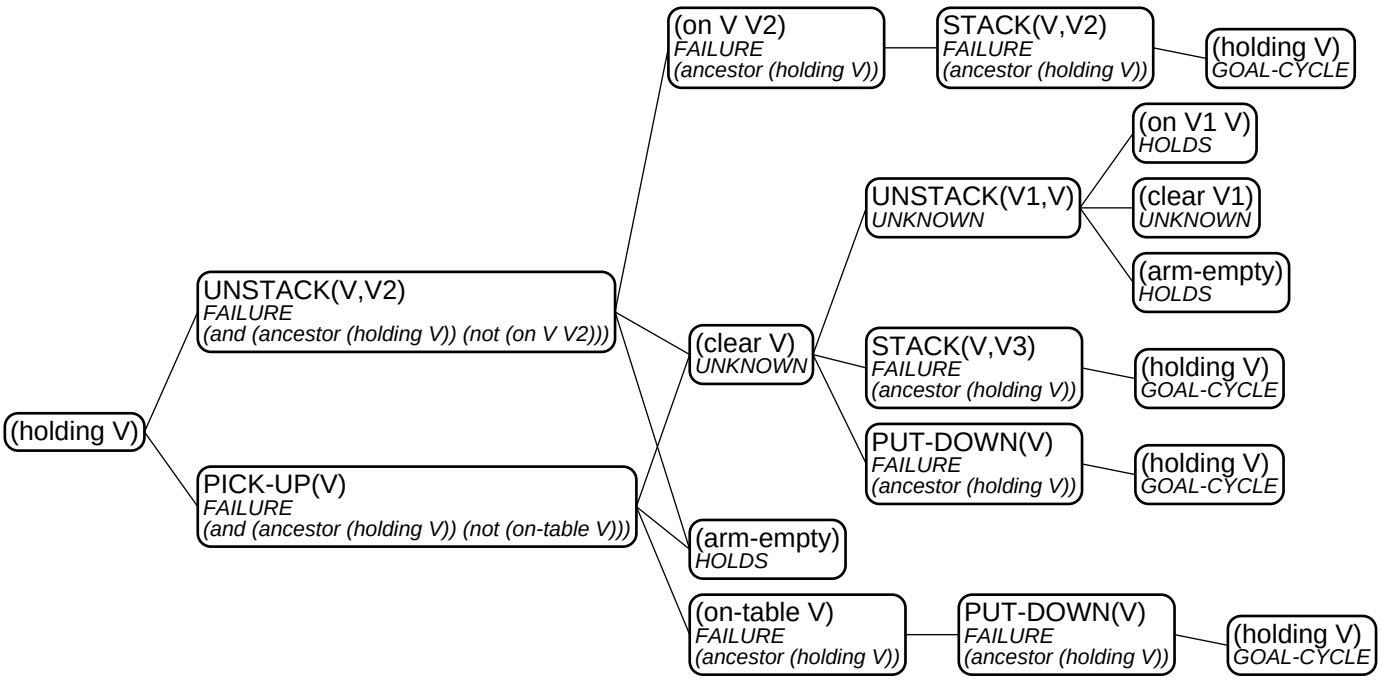

Figure 2: The annotated `holding` PSG.

3.4 Extracting Control Rules from the PSG

STATIC traverses each labeled PSG, in preorder, searching for operator nodes labeled *failure* or *success*. Operator rejection and binding selection rules are learned from *failure* nodes, and preference rules are learned from *success* nodes. No rules are learned from nodes labeled *unknown*. The operator rejection rules extracted from the `holding` PSG, for example, appear in Table 9. The first rule, which rejects the operator PICK-UP, is derived from the PICK-UP(V) node in the PSG; the second rule is derived from the UNSTACK(V,V2) node in the PSG. To avoid learning redundant rules, once STATIC encounters a node labeled *failure*, it does not continue its traversal of the PSG below that node. A detailed specification of STATIC's criteria for generating control rules, and minor extensions to the algorithms presented above (e.g. forming node rejection rules, avoiding redundant rules) are described in [13].

```
(REJECT-PICK-UP
  (if (and (current-node Node)
           (current-goal Node (holding Block-X))
           (candidate-op Node pick-up)
           (known Node (not (on-table Block-X)))))
  (then (reject operator pick-up)))

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

Table 9: The operator rejection rules extracted by STATIC from the `holding` PSG in Figure 2.

3.5 Complexity Analysis

STATIC constructs a PSG in one pass, annotates it with labels and failure conditions in a second pass, and extracts control rules in a third pass. Thus, its running time is "essentially linear" in the size and number of its PSGs. There are three caveats to this statement. First, STATIC's detection of shared subgoals during PSG construction requires inspecting all the sibling operators at each subgoal node. Second, STATIC's detection of goal-cycles requires inspecting the ancestor list at each subgoal node. Finally, STATIC constructs failure conditions at nodes labeled *failure*. In the worst case, the maximal size of a failure condition is the size of the PSG, making STATIC's running time quadratic in PSG size. In practice, however, failure conditions tend to have bounded size keeping STATIC's running time linear in PSG size.

More formally, let η denote the number of nodes in the PSGs; let β denote the maximal number of operators relevant to achieving a subgoal; let α denote the maximal number of preconditions to an operator; let ϕ denote the number of predicates in the PSG; and let δ denote the maximal failure condition size. The running time of the algorithm described above is $O(\eta\beta\alpha^2\phi\delta)$.

4 Static Analysis of Goal Interactions

Goal interactions such as goal clobbering and prerequisite violation cause PRODIGY to backtrack. Goal clobbering occurs when, in the process of achieving one goal, the problem solver negates or *clobbers* a goal that is already satisfied. In the Schedworld, for example,

changing the shape of an object damages any polish on the object's surface. Thus, the goal `(surface-condition Obj polished)` is clobbered by all plans that achieve the goal `(shape Obj cylindrical)`. Prerequisite violation occurs when, in the process of achieving one goal, the problem solver renders a pending goal unachievable by clobbering one of its prerequisites. In the Schedworld, making an object cylindrical by using the ROLL operator has the side-effect of heating the object. An object is only clampable if it is cold, and there is no way of cooling an object. Consequently, any goal that has a clamping prerequisite is rendered unachievable by the ROLL operator.

STATIC anticipates these goal interactions by analyzing its PSGs, seeking to determine the side effects and vital prerequisites of various goals. Based on this analysis, STATIC generates goal ordering and operator preference control rules that enable PRODIGY to avoid goal interactions (e.g. Table 10). 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 these constraints. An alternative optimization strategy (taken by Knoblock's ALPINE [24]) is to anticipate all goal interactions that can *possibly* occur. Section 6.2 analyzes the relationship between the two approaches. STATIC is *incomplete* in that it will not anticipate all goal interactions, but it is *sound* in that the interactions that STATIC anticipates would in fact occur, if it were not for STATIC's control rules.

```
(PREFER-SHAPE
  (if (and (current-node Node)
           (candidate-goal Node (shape Part-X cylindrical))
           (candidate-goal Node (surface-condition Part-X polished))))
  (then (prefer goal (shape Part-X cylindrical)
                     (surface-condition Part-X polished))))
```

Table 10: A goal ordering rule produced by STATIC.

The remainder of this section is organized as follows. Section 4.1 defines precisely several terms used in the discussion that follows. Section 4.2 discusses goal clobbering followed by Section 4.3 on prerequisite violation. Both sections:

- Define the interaction formally.
- Present STATIC's algorithm for anticipating the interaction.
- Prove the algorithm to be sound.
- Describe STATIC's strategy for deriving control rules.
- Analyze the time and space complexity of the algorithm.

4.1 Preliminaries

A variable substitution σ is a function from logical formulas to logical formulas that replaces variables by constants (or variables). If σ maps X to a, for example, then σ((on X Y))=(on a Y). A conjunction S_1 is said to *match* a conjunction S_2 when there is some substitution σ that makes every literal in S_1 identical to some literal in S_2. That is, $\forall s_1 \in S_1, \exists s_2 \in S_2$, such that $\sigma(s_1) = s_2$. Thus, `(on X b)` is said to match `(on-table b)` `(on a b)` `(clear a)` when σ maps X to a. The notation $x|y$ means that x matches y; $\neg x|y$ means that x does not match y. The empty conjunction, denoted by ϕ, matches any conjunction. Thus, $\forall x$ we have that $\phi|x$. Finally, $\forall x$ s.t. $x \neq \phi$, we have that $\neg x|\phi$.

A literal g_1 is said to *negate* a second literal g_2 when both literals cannot match the same state, under some substitution σ. I denote this fact by $N(g_1, g_2, \sigma)$.

$$N(g_1, g_2, \sigma) \equiv \sigma(g_1)|s \rightarrow \neg\sigma(g_2)|s.$$

Note that the negation relation is symmetric.

Under what conditions does one literal negate another? Clearly, for any σ, it is the case that $N(g_1, \neg g_1, \sigma)$. In addition, STATIC makes use of explicit axioms that state which goals negate each other in a given problem space. In the Schedworld, for example, a Part-X is either `hot` or `cold`. STATIC is told, therefore, that `(temperature Obj hot)` negates `(temperature Obj cold)`. STATIC's "goal negation" axioms for the Schedworld appear in Table 11.

Determining whether one literal necessarily negates another turns out to be easy. To check whether one literal negates another, under some variable substitution, STATIC matches the two literals against the literals in each of the goal negation axioms. The matching procedure either signals failure, implying that literals do not negate each other, or returns a (possibly empty) variable substitution implying that the literals negate each other under that substitution.

$\forall\ Obj, V, W$ such that $V \neq W$:

$$temperature(Obj, hot) \rightarrow \neg temperature(Obj, cold)$$
$$painted(Obj, V) \rightarrow \neg painted(Obj, W)$$
$$\text{surface-condition}(Obj, V) \rightarrow \neg\text{surface-condition}(Obj, W)$$
$$shape(Obj, V) \rightarrow \neg shape(Obj, W)$$

Table 11: Schedworld goal negation axioms.

4.2 Goal Clobbering

A plan for achieving a goal g is said to *clobber* a protected goal p when p is negated by the plan. A goal g is said to *necessarily* clobber a protected goal p when *all* plans for achieving

g clobber p. This section defines the notion of necessary goal clobbering, and shows how STATIC anticipates and avoids goal clobbering by computing the necessary effects of achieving each goal.

Typically, g clobbers p onlys under some co-designation constraint between the two goals. In our Schedworld example, goal clobbering occurs only when the `surface-condition` and the `shape` goals refer to the same object. In the discussion that follows, co-designation constraints are enforced by applying the variable substitution σ to both goals. The set of states in which the clobbering necessarily occurs may be constrained as well. In the Schedworld, two operators are available for achieving the goal `has-hole`: PUNCH and DRILL-PRESS. Only PUNCH clobbers the `surface-condition` goal. If the object in question is not drillable, however, the only available operator is PUNCH and the goal clobbering will necessarily occur. Constraints on the set of states are captured by the condition k described below.

Formally, a goal g is said to necessarily clobber a protected goal p, under some condition k and variable substitution σ, when every nonempty operator sequence c that achieves g, from a state that matches p and k under σ, has a prefix that results in a state that does not match p under σ. The necessary clobbering relation is denoted by $\Box C(g, p, \sigma, k)$.

$$\Box C(g,p,\sigma,k) \equiv \forall s \text{ s.t. } \sigma(p)|s \wedge \sigma(k)|s, \text{ and } \forall c \text{ s.t. } \sigma(g)|c(s), \\ \exists c' \text{ such that } \mathit{prefix}(c', c), \text{ and we have that } \neg\sigma(p)|c'(s).$$

Let $E(g)$ denote the necessary effects of achieving g. $E(g)$ is defined to be a set of pairs (e, k) where e is an effect and k is a condition under which e necessarily occurs.

$$E(g) = \{(e,k)|\ \forall s, \sigma \text{ s.t. } \sigma(k)|s, \text{ and } \forall c \text{ s.t. } \sigma(g)|c(s), \\ \exists c' \text{ such that } \mathit{prefix}(c', c), \text{ and we have that } \sigma(e)|c'(s)\}$$

g clobbers a protected goal p if and only if p is negated as a necessary effect of achieving g.

Proposition 2 *$\Box C(g, p, \sigma, k)$ iff $\exists (e, k) \in E(g)$, s.t. $N(e, p, \sigma)$.*

Proof: By the definitions of $\Box C$, E, and N. □

Note that operator sequences that clobber but re-establish p in the process of achieving g satisfy $\Box C(g, p, \sigma, k)$. STATIC detects and attempts to avoid "temporary goal clobbering" of this sort for two reasons. First, temporary goal clobbering can result in a state cycle, which causes PRODIGY to backtrack. Second, in order to eliminate "temporary effects" from consideration STATIC would have to determine the *possible* side-effects of achieving each subgoal. While this can be done in a straight-forward manner, using an algorithm such as Knoblock's [24], I did not pursue this in STATIC.

4.2.1 Computing Necessary Effects

I conjecture that computing the set of necessary effects E is hard in general, but I will not attempt to prove this. STATIC computes a subset of E denoted by $\hat{E}$, and anticipates

goal clobbering based on $\hat{E}$. This section describes how $\hat{E}$ is computed and discusses the relationship between E and $\hat{E}$.

STATIC computes $\hat{E}$ using two mutually recursive procedures, shown in Table 12. The first procedure takes two arguments: a goal g and its ancestor set G.[4] The procedure intersects (a subset of) the necessary effects of each operator that could potentially achieve g. The subset is denoted by $\hat{E}_o(g, G)$ for any given operator o. $\hat{E}_o(g, G)$ is computed by a second procedure which forms the union of o's effects with the necessary effects of its preconditions. The necessary effects of each precondition are computed by calling the first procedure recursively. Note that a necessary effect of achieving a precondition occurs only if the precondition, and each of its ancestors, is unmatched in the current state. Otherwise, PRODIGY would not attempt to achieve the precondition. Thus, each necessary effect is conditioned on the negation of its ancestors. Let $\bar{G}$ denote the conjunction of the negations of the literals in the set of ancestor literals G. That is, $\bar{G} \equiv \bigwedge_{g \in G} \neg g$. Table 12 denotes an effect e, which occurs only in the context of G, by the pair $(e, \bar{G})$.

$U(g, G)$ is satisfied when g cannot be achieved in the context of G. $holds(g, G)$ is satisfied when g is necessarily true in the context of G (see Section 3.2.1). $leaf(g)$ is satisfied when g is a leaf of the PSG.

$$\hat{E}(g, G) = \text{If } U(g, G) \text{ or } holds(g, G) \text{ then return } \phi.$$
$$\text{Else if } leaf(g) \text{ then return } (g, \bar{G})$$
$$\text{Else return } \bigcap_{o \in ops(g)} \hat{E}_o(g, G + g)$$

Intersecting effect pairs:

$$(e_1, k_1) \cap (e_2, k_2) = \begin{cases} (\sigma(e_1), \sigma(k_1) \wedge \sigma(k_2)) & \text{when } \exists \sigma \text{ such that } \sigma(e_1) = e_2. \\ (\sigma(e_2), \sigma(k_1) \wedge \sigma(k_2)) & \text{when } \exists \sigma \text{ such that } \sigma(e_2) = e_1. \\ \phi & \text{otherwise.} \end{cases}$$

$$\hat{E}_o(g, G) = \{(e, \bar{G}) | e \in effects(o)\} \cup \left(\bigcup_{p \in precs(o)} \hat{E}(p, G) \right)$$

Table 12: The procedure for computing $\hat{E}$(g,G).

As Table 12 indicates, a necessary effect of an operator that achieves g is only a necessary effect of the subgoal g if it is shared by all the operators that could potentially achieve g.

[4] When g is a top-level goal, $G = \phi$.

The shared effects are determined by intersecting the necessary effect sets of each operator. Intersecting elements of the sets involves matching the individual effects of operators (Table 12). Even if the effects of two operators share a predicate they may not match when the arguments to the predicate are bound differently in each operator. STATIC's matching procedure takes this into account by distinguishing between four classes of bindings for an argument. Consider matching the effect, $e(X)$, of one operator against the effect $e(Y)$ taken from another operator. The arguments X and Y can be:

- Bound to a goal argument: in this case $X = Y$ and the effects match.
- Bound to operator-specific constants in matching the operator's effects against the goal: when the operator-specific constants are distinct (e.g., $X = b$ and $Y = c$), the effects will not match.
- Bound in matching each operator's preconditions against the state: the effects match only when STATIC is able to identify constraints under which the effects are necessarily the same. For example, suppose the effects are `(dr-open D-X)` and `(dr-open D-Y)`. Then, the effects match when both `D-X` and `D-Y` are doors to the same room, and the room only has one door.
- Unbound: If either X or Y is unbound, the effects will match. An argument to an effect is unbound when it does not appear in the operator's preconditions and the effect is not matched against the goal. Unbound arguments are marked as such when the PSG is constructed.

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-condition Obj polished)`. STATIC detects this necessary side-effect by intersecting $\hat{E}_{lathe}$ and $\hat{E}_{roll}$.

Since the effects of o are added to $\hat{E}_o$ after the effects of o's preconditions, the list $\hat{E}_o(g, G)$ is ordered by the generality of the conditions on o's necessary effects. This ordering turns out to be useful because STATIC searches $\hat{E}$ sequentially for clobbering effects. As a result, STATIC will form the most general rules it can find.

The computation of necessary effects by STATIC is sound:

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

Proof: (By induction on the number of calls to $\hat{E}$.) □

It is difficult characterize the relationship between $\hat{E}$ and E in general terms. In the problem spaces studied, $\hat{E}$ contained enough information to allow STATIC to anticipate virtually all binary goal interactions encountered by PRODIGY. Yet it is possible to manufacture examples where $\hat{E}$ is strictly a subset of E. For instance, any necessary effects resulting from recursive operator application will be missed by STATIC. Furthermore, consider a PSG where some goal g can be achieved by the operators o_1 and o_2. Suppose that STATIC is unable to

identify the condition f under which o_1 fails.[5] Suppose there are only two operators for achieving g. It follows that, when f holds, all of the necessary effects of o_2 are necessary effects of g, but STATIC will miss this.

It follows that by computing $\hat{E}$ we can anticipate some (but not all) goal clobberings.

Corollary 1 *If* $\exists(e,k) \in \hat{E}(g,\phi)$, $\exists\sigma$ *s.t.* $N(e,p,\sigma)$ *then* $\Box C(g,p,\sigma,k)$.

Proof: By Proposition 2 and Theorem 1. □

4.2.2 Avoiding Goal Clobbering

STATIC seeks to avoid goal clobbering using its knowledge of necessary effects. For every goal pair (p,g), STATIC searches for a necessary effect of achieving g, that clobbers p under some variable substitution σ. If a clobbering is found, and the two goals do not negate each other under σ, STATIC forms a goal ordering rule in order to avoid the clobbering. A sample goal ordering rule appears in Table 10.

As Ryu and Irani [49] demonstrate, introducing the appropriate co-designation on the variables in p and g can be rather subtle. Consider Ryu and Irani's Stripsworld example where p is `(dr-closed D-X)` and g is `(inroom robot R-Y)`. The `inroom` goal has the necessary effect `(dr-open D-Y)` where `D-Y` is a door to room `R-Y`. The `(dr-open D-Y)` effect necessarily clobbers the `(dr-closed D-X)` goal only when the two variables `D-X` and `D-Y` necessarily co-designate. To generate the appropriate goal-ordering rule, STATIC adds a condition to the rule which guarantees that the two variables necessarily co-designate (Table 13). In essence, the condition ensures that `R-Y` only has one door, in which case achieving the `inroom` goal necessarily clobbers the `dr-closed` goal.[6]

```
(PREFER-INROOM-ROBOT
 (if (and (current-node Node)
          (candidate-goal Node (inroom robot R-Y))
          (candidate-goal Node (dr-closed D-X))
          (forall (D-Y) (known Node (dr-to-rm D-Y R-Y))
                        (is-equal D-Y D-X))))
 (then (prefer goal (inroom robot R-Y)
                    (dr-closed D-X)))
```

Table 13: A STATIC goal ordering rule enforcing co-designation constraints.

[5] For instance, o_1 may lead to a state cycle.

[6] A bug prevented STATIC from generating this condition in the STATIC experiments described earlier, leading STATIC to produce an overly-general rule. This bug has been corrected in response to Ryu and Irani's analysis.

When the above procedure fails to find goal ordering rules, STATIC checks whether one or more of the operators that achieve g clobbers p. If so, then when all other operators for achieving g fail, a goal clobbering will occur. When all non-clobbering operators have failure labels, STATIC creates an expression, called the goal-clobbering condition, which consists of the conjunction of the failure conditions of each of the non-clobbering operators. g necessarily clobbers p when the goal-clobbering condition holds.

Suppose, for example, that PRODIGY tries to achieve the two Schedworld goals: `(has-hole Obj Width Orientation)` and `(surface-condition Obj polished)`, and it turns out that `Obj` is not drillable in the desired orientation. It follows that PRODIGY has to use the operator PUNCH to achieve `has-hole`. The PUNCH operator clobbers the `(surface-condition Obj polished)` goal. STATIC generates a rule that tells PRODIGY to achieve `(has-hole Obj Width Orientation)` before `(surface-condition Obj polished)` when `Obj` is not drillable (Table 14) in order to avoid the goal clobbering. Finally, when a goal-clobbering condition cannot be found, STATIC tells PRODIGY to prefer other operators to the clobbering ones.

```
(PREFER-HAS-HOLE
  (if (and (current-node Node)
           (candidate-goal Node (has-hole Part-X Width-W Orient-Z))
           (candidate-goal Node (surface-condition Part-X polished))
           (known Node (not (is-drillable Part Orient-Z)))))
  (then (prefer goal (has-hole Part-X Width-W Orient-Z)
                     (surface-condition Part-X polished))))
```

Table 14: A "conditional" goal ordering rule produced by STATIC.

Determining the running time of STATIC's algorithm is straight forward. As shown in Table 12, the necessary effects of achieving a goal are computed by traversing the PSG rooted in that goal. To detect goal clobbering, STATIC considers every pair of (uninstantiated) achievable literals. Thus, its running time is linear in the size of the PSGs, and quadratic in the number of achievable literals. Determining whether two literals negate each other is linear in the number of goal negation axioms. Finally, the operations on sets of necessary effects take time linear in the size of the relevant $\hat{E}$s.

4.3 Prerequisite Violation

A plan for achieving a goal v is said to *violate the prerequisites* of a goal g if executing the plan for achieving v renders g unachievable. This section defines the notion of necessary prerequisite violation, and shows how STATIC anticipates and avoids prerequisite violation by computing the necessary prerequisites of achieving each goal.

A goal v is said to necessarily violate the prerequisites of a pending goal g, under some condition k and variable substitution σ, when every operator sequence that achieves v, from any state that matches k under σ, renders g unachievable under σ. The violation relation is denoted by $\Box V(g, v, \sigma, k)$.

$$\Box V(g, v, \sigma, k) \equiv \forall s \text{ s.t. } \sigma(k)|s \text{ and } \forall c \text{ s.t. } \sigma(v)|c(s), \neg\exists c' \text{ s.t. } \sigma(g)|c'(c(s)).$$

Note that STATIC ignores ignores "soft" prerequisite violations–ones that can be undone by inserting additional operators into the plan. Let $P(g)$ denote the necessary prerequisites of achieving g. Since some prerequisites are only necessary if a condition k matches s, $P(g)$ is defined to be a set of pairs (p, k) where p is a prerequisite and k is a condition under which p is necessary.

$$P(g) = \{(p, k)|\forall s, \sigma \text{ s.t. } \sigma(k)|s, \text{ and } \forall c \text{ s.t. } \sigma(g)|c(s) \text{ we have that } \sigma(p)|s\}$$

v violates the prerequisites of g if and only if a necessary prerequisite of a goal g is negated as a necessary effect of achieving the goal v.

Proposition 3 *$\Box V(g, v, \sigma, k)$ iff $\exists (p, k) \in P(g)$, $\exists (e, k) \in E(v)$ such that $N(e, p, \sigma)$.*

Proof: By the definitions of $P, E, \Box V$, and N. □

4.3.1 Computing Necessary Prerequisites

I conjecture that computing the set of necessary prerequisites P is hard in general, but I will not attempt to prove this. STATIC computes a subset of P denoted by $\hat{P}$, and identifies prerequisite violations based on $\hat{P}$. This section describes how $\hat{P}$ is computed.

The procedure for computing $\hat{P}$ takes two arguments: a goal g, and g's ancestor set G. The procedure calls two Boolean-valued functions: $recurs(g, G)$ which returns true when g's predicate appears in G, and $holds(g, G)$ which returns true if g is necessarily matched by the current state given G and the constraints on legal states. The procedure appears in Table 15.

The computation of necessary prerequisites by STATIC is sound:

Theorem 2 *$\hat{P}(g, \phi) \subseteq P(g)$.*

Proof: (By induction on the number of calls to $\hat{P}(g, G)$.) □

Again, it is difficult to characterize the relationship between $\hat{P}$ and P more precisely. As with $\hat{E}$ and E, there are cases where $\hat{P}$ is a strict subset of P, yet the procedure described performs well in practice.

It follows that computing $\hat{P}$ enables STATIC to anticipate some (but not all) prerequisite violations.

Corollary 2 *If $\exists (p, k) \in \hat{P}(g, \phi)$, $\exists (e, k) \in \hat{E}(v, \phi)$, $\exists \sigma$ s.t. $N(e, p, \sigma)$ then $\Box V(g, v, \sigma, k)$.*

Proof: By Theorem 1, Theorem 2, and Proposition 3. □

$\hat{P}(g, G) =$ If recurs(g,G) or holds(g,G) then return ϕ
Else if $U(g, G)$ then return $(g, \bar{G})$
Else return $\bigcap_{o \in ops(g)} \hat{P}_o(g, G + g)$

$$\hat{P}_o(g, G) = \bigcup_{p \in precs(o)} \hat{P}(p, G)$$

Table 15: The procedures for computing $\hat{P}$.

4.3.2 Avoiding Prerequisite Violation

STATIC attempts to avoid prerequisite violations using its knowledge of necessary prerequisites and effects. STATIC's strategy for avoiding prerequisite violations is similar to its strategy for avoiding goal clobbering. Essentially, for every goal pair (v, g) STATIC applies Corollary 2 to anticipate necessary prerequisite violations and forms goal reordering and operator preference rules to avoid the violations.

The time and space complexity of detecting and avoiding prerequisite violation are the same as those for detecting and avoiding goal clobbering. If goal clobbering is analyzed first, the necessary effects can be cached and used in analyzing prerequisite violation. In fact, the necessary prerequisites of a PSG node can be determined by examining the node's failure condition (see Table 8). Thus, prerequisite violation can be avoided solely on the basis of information that has already been computed by analyzing failure and goal clobbering.

4.4 Discussion

STATIC's analysis of goal interactions embodies several high-level design choices:

- *Forming control rules based on necessary (as opposed to possible) goal interactions.* This design choice was made to emulate PRODIGY/EBL's strategy for learning from goal interactions.

- *Analyzing interactions between goal pairs as opposed to N-ary goal interactions for some N greater than two.* This design choice was made to keep STATIC's analysis process tractable. In practice, more involved goal interactions appear to be rare, difficult to analyze at compile-time, and expensive to anticipate at run time.

- *Searching for "unconditional" goal interactions before conditional ones.* In many cases goal interactions have multiple explanations. This design choice leads STATIC to prefer explanations that do not depend on any facts about the state. As a result, the

match cost and coverage of STATIC's goal ordering rules are improved as compared with PRODIGY/EBL's.

STATIC's analysis can be used to avoid goal interactions, not only in PRODIGY, but in subgoal-interleaving or forward-chaining planners as well. In the Schedworld, for example, STATIC notes that changing the shape of a Part-X destroys the Part-X's surface polish, and recommends changing shape before polishing. This recommendation is useful independent of PRODIGY. *Any* planner applied to the Schedworld problem space ought to generate plans that change the shape of a Part-X before polishing it. Thus, unless the planner anticipates goal interactions on its own, it will find STATIC's analysis of goal interactions helpful. Work is currently underway to apply STATIC-style PSG analysis to the UCPOP partial-order planner [43] and to a new subgoal-interleaving variant of PRODIGY [3].

5 Comparing STATIC and PRODIGY/EBL

This section answers the fundamental questions posed in the introduction by comparing STATIC and PRODIGY/EBL along two dimensions:

- **Impact:** the ability to speed up PRODIGY.
- **Cost:** the time required to generate control knowledge.

Surprisingly, STATIC outperforms PRODIGY/EBL along both dimensions in the problem spaces studied.[7] This section analyzes STATIC's success, and considers how the relative performance of the two systems will scale up (Section 5.3), noting cases in which PRODIGY/EBL will outperform STATIC (Section 5.3.3).

I refer to the PRODIGY problem solver, running under the guidance of STATIC's control rules, as PRODIGY+STATIC and to PRODIGY, guided by the rules learned by PRODIGY/EBL, as PRODIGY+EBL. The comparisons use CPU-time (on a Sun Sparcstation) as a performance measure. A more detailed comparison that considers the number of nodes, average match cost, and number of rules learned by each of the systems appears in [13, chapter 8]

5.1 Impact

STATIC speeds up PRODIGY considerably in the problem spaces studied (Table 16). In fact, STATIC's control knowledge is up to three times as effective as that learned by PRODIGY/EBL. While the exact figures depend on a number of factors such as PRODIGY/EBL's training procedure and the problem distribution used in the experiments, the response to the first fundamental question is clear: static analysis *can* produce effective control knowledge.

[7] 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.

	PRODIGY	PRODIGY+EBL	PRODIGY+STATIC	Ratio
Blocksworld	2182	139	47	2.96
Stripsworld	4347	292	226	1.29
Schedworld	4391	1262	685	1.84
ABworld	869	925	711	1.3

Table 16: Total problem-solving time in CPU seconds. The rightmost column displays the ratio of PRODIGY+EBL's time to that of PRODIGY+STATIC.

How was STATIC able to outperform PRODIGY/EBL despite STATIC's narrower scope (e.g. no recursive explanations) and weaker inputs[8] (i.e., no training examples)? This question is addressed in Section 7.

5.2 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 from twenty-six to four-hundred and sixty times faster than PRODIGY/EBL (Table 17). Thus, in response to the second fundamental question: static analysis can produce effective control knowledge *quickly.*

	PRODIGY/EBL	STATIC	Ratio
Blocksworld	762	9.9	77.0
Stripsworld	1057	40.5	26.1
Schedworld	1374	19.4	70.8
ABworld	11,233	24.4	460.4

Table 17: Learning time in CPU seconds.

PRODIGY/EBL was "trained" by following the learning procedure outlined in Minton's dissertation. First, PRODIGY/EBL analyzed a set of randomly generated training problems (approximately one hundred in each problem space). Second, PRODIGY/EBL estimated the utility of the rules learned in the first phase using an additional set of randomly generated problems (roughly twenty-five in each space). Table 18 decomposes PRODIGY/EBL's running time into three components: time to solve training problems, time to construct proofs based on the training-problem traces, and time to perform utility evaluation. It is interesting to note that utility evaluation is quite costly.

[8]Both PRODIGY/EBL and STATIC utilize domain-specific axioms to generate control rules. A detailed comparison shows that the two systems are given roughly the same knowledge but in somewhat different forms [13, chapter 8].

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

Table 18: PRODIGY/EBL's learning time decomposed into components: time to solve training problems, 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 problems. 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, its 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 19 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 19: 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 problems, and PSG size remains large. The following factors contribute to keeping PSGs more compact than problem-solving traces in general:

- PSGs are partially instantiated. Thus, PSG size remains constant as the number of objects in the problem solver's state increases whereas trace size increases commensurately.
- PSGs do not represent recursions whereas traces do.
- PSGs do not represent backtracking due to goal interactions whereas traces do.

5.3 Scaling Up

STATIC's success is due, in part, to the nature of the problem spaces studied. However,

with the exception of the ABworld, the problem spaces studied were not selected to highlight STATIC's performance. In fact, the problem spaces were chosen by Minton to test PRODIGY/EBL's effectiveness [30], and have been used as benchmarks in a number of experiments both inside and outside the PRODIGY group [2, 18, 24]. Thus, STATIC has been tested as rigorously as PRODIGY/EBL, and other speedup learning programs. Naturally, additional experiments in more realistic problem spaces are needed to further test the entire gamut of speedup learning programs. The theoretical analysis of STATIC, below, provides additional insight into STATIC's performance relative to PRODIGY/EBL. The analysis considers the relative running time and scope of the two systems, describing several cases in which PRODIGY/EBL is likely to outperform STATIC (Section 5.3.3).

5.3.1 Complexity Analysis

The analysis of STATIC's algorithms in Sections 3.1 and 4 shows that STATIC's running time is:

- *Linear in the number of possible subgoals*, each of which forms the root of a distinct PSG. Note that modifying STATIC to only create the PSGs whose roots are subgoals that arise in actual problems, will make STATIC's complexity the same as PRODIGY/EBL's, on this measure. This simple modification would help STATIC in problem spaces with large number of possible subgoals, only a few of which are ever encountered in practice.

- *Polynomial in the number of operators relevant to each subgoal.* Since proving the failure of a subgoal requires analyzing all of the relevant operators, PRODIGY/EBL's complexity is the same as STATIC's for this measure as well.

- *Exponential in the depth of nonrecursive subgoaling in the problem space.* Since PSG expansion is terminated at a literal node when that literal's predicate is identical to the predicate of one of its ancestors, PSG depth is bounded by the number of predicates in the problem space. In practice, actual PSG depth is substantially smaller since many predicates do not appear in a single subgoaling sequence. Since PSG size is exponential in PSG depth, the depth of nonrecursive subgoaling is the key to the tractability of STATIC's analysis. STATIC's analysis will be intractable in any problem space that exhibits deep nonrecursive subgoaling, and nontrivial branching.

5.3.2 Scope

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, but this may not be the case in other problem spaces.

5.3.3 Worst Case Scenarios for STATIC

Two worst-case scenarios have already been mentioned above. First, when a combination of deep PSGs and shallow problem-solving traces is encountered, STATIC will be significantly slower than PRODIGY/EBL. Second, STATIC will fail when the analysis required to curtail problem solving falls outside its scope. In addition, when proving both failure and success is recursive as in the Tower of Hanoi, ABworld, or recursive Bin-world [19] problem spaces, STATIC will form trivial PSGs and refuse to generate much if any control knowledge. To its credit, STATIC will do so quickly, whereas EBL systems will often generate ineffective control knowledge in such spaces, and take a long time to do so [13, 19, 23].

As EBL lore would have it, PE is bound to be intractable when applied to sufficiently large and complex theories. The analysis in this section shows that when PE is appropriately constrained, as in STATIC, PE will only be intractable, relative to EBL, when potential nonrecursive subgoaling in the problem space is "deep" and actual problem solving is "shallow." In Minton's experiments the opposite was true: nonrecursive subgoaling was shallow whereas, due to recursion, actual problem solving was relatively deep. Whether such cases are prevalent in the real world is, ultimately, an empirical question that cannot be prejudged.

6 Related Work

Problem spaces can be viewed as programs. This perspective places STATIC and EBL in the broader context of program optimization. Much of the classical work on optimizing compilers (see [1]) focuses on optimizing arithmetic operations in programs written in imperative languages and is not relevant for this reason. Similarly, much of the work on optimizing database queries is specific to relational algebra, and as Smith points out [55], many of the optimizations of relational algebra queries are obtained automatically as a side effect of the partial instantiation of conjunctive expressions that is performed by search "engines" such as Prolog and PRODIGY. Furthermore, whereas relational algebra does not allow recursion, the optimization of recursive problem spaces is a central issue here.

The remaining literature on program optimization is vast, and cannot be surveyed exhaustively here.[9] Instead, I define a space of program optimization systems, position STATIC in this space, and contrast it with other systems along the different dimensions. I formulate the dimensions of the space as questions:

- *How does the optimizer handle recursion?* STATIC engages in partial problem space analysis, terminating the expansion of the PSG whenever recursion is encountered . In

[9] See [41, 57] for extensive surveys.

contrast, much of the research on partial evaluation is concerned with effective policies for unfolding recursions (e.g. [58]).

- *Is the optimizer fully automatic?* STATIC is automatic but many optimizers are not (e.g. [9]).

- *What program transformations does the optimizer employ?* STATIC employs unfolding and reordering transformations. In contrast to most Prolog implementations of EBL (e.g., [21, 52, 56]), which only reorder Horn Clauses, STATIC (and PRODIGY/EBL) reorder both Horn clauses and conjuncts.

  Conjunct ordering has been studied extensively in the database community (e.g. [60]) and by Smith [54]. By and large, the database work focuses on heuristics that order conjunctive subgoals based on factors such as domain size and number of instantiated arguments. Additional heuristics are used to decompose queries into subqueries that can be processed independently. Unlike PRODIGY/EBL and STATIC, the database systems do not perform conjunct reordering in concert with unfolding.

- *What representation of the program does the optimizer use?* STATIC uses Problem Space Graphs (PSGs). Many optimizers use similar graph representations of programs. Connection graphs [26], rule/goal graphs [57], and flow graphs [38] are some examples.

- *What is the complexity of the optimization?* STATIC's time and space complexity are close to linear in the size of its PSGs. The complexities of different optimizers vary widely.

- *Are the optimizations carried out for specific inputs?* STATIC is run once per problem space. Many approaches, particularly ones cast as improved proof procedures (e.g., [26]) are problem specific. Most optimizers that analyze goal interactions are problem specific as well (see Section 6.3).

- *Does the optimizer change the semantics of the program?* Most optimizers, STATIC included, use only semantics-preserving transformations.

6.1 STATIC and Traditional PE

Although STATIC utilizes PE intrinsically, it goes beyond standard partial evaluators in several important ways that are motivated by its task of generating search-control knowledge for PRODIGY. First and foremost, STATIC utilizes (its own versions of) PRODIGY/EBL's multiple target concepts. The concepts facilitate a novel solution to a major open problem for PE: deciding when to terminate the unfolding of recursive calls [51, 58]. STATIC avoids unfolding recursive calls altogether, relying on its "nonstandard" target concepts to yield effective control knowledge despite its strong termination criterion. In addition, STATIC expresses its output as control rules, which encode reordering transformations that are not

part of the standard arsenal of PE systems. Finally, STATIC anticipates goal clobbering and prerequisite violation whereas standard PE systems do not.

In AI work, PE has been used to generate abstraction hierarchies for planning [7] and to reduce the match cost of rules learned via EBL [47, 29]. Concurrently with STATIC, Letovsky [28] developed the PROPE system which generates macro-clauses by partially-evaluating pure PROLOG programs. Letovsky reports that PROPE failed to generate the appropriate search-control knowledge for solving simple algebraic equations. PROPE is weaker than STATIC in a number of ways. PROPE does not generate control rules, does not analyze failures and goal interactions, and does not use axioms to prune "spurious execution paths" (cf. Section 3.2.1).

6.2 Analyzing Goal Interactions

Goal interactions occur frequently in programs that attempt to achieve conjunctive goals by nonmonotonic state transformations. The program transformations used to avoid goal interactions are the reordering transformations used to optimize monotonic inference programs, but the motivation for applying these transformations is different.

REFLECT Dawson and Siklossy [10] 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, and 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)` and `(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.

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 (see Section 3.1), and particularly its avoidance of recursion. Another advance is the formal specification and computation of necessary effects and prerequisites to facilitate goal reordering (Section 4). Finally, unlike STATIC, REFLECT does not reorder subgoals to avoid goal interactions.

ALPINE Knoblock's ALPINE [24] 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. ALPINE's algorithm guarantees that if two literals potentially interact in the problem space, then the algorithm will place an edge between them. The algorithm is *conservative* in that the algorithm will sometimes

include edges between literals that do not interact. Knoblock's guarantees on the algorithm'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.

Universal Plans Schoppers [50] 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 problem-space definitions 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 considering tightly circumscribed domains such as the 3-block Blocksworld where recursion is bounded and variables are fully instantiated. Instead, Schoppers focuses on different issues including conditional effects and incorrect world models.

6.3 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 [4] 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 [5] describe an algorithm that, given an initial state and a goal conjunction, computes a partial order on the subgoals in the goal conjunction, which obviates backtracking across subgoals. The algorithm runs in $O(n^3)$ time where n is the number of subgoals. Ryu and Irani [49] describe an algorithm that relies on training examples to analyze goal interactions. The algorithm was tested in PRODIGY's benchmark problem spaces and shown to generate goal-ordering rules that are at least as effective as STATIC's rules. Finally, Knoblock's ALPINE, which was discussed above, performs some initial preprocessing of the problem space definition, but then develops considerably stronger constraints on possible goal interactions by analyzing individual problems.

In the context of logic programming, optimization is often implemented via improved proof procedures, some of which use graph representations of their rule sets similar to STATIC's PSGs. Connection graph proof procedures, for example, rely on graphs whose nodes represent rules or facts and whose edges represent the most general unifiers between the adjacent nodes [26]. Although some of the optimizations implemented by connection graph proof procedures are akin to STATIC's, the proof procedures are query-specific, yielding very different graph-traversal algorithms. In addition, the proof procedures do not seek

to anticipate goal interactions, a central focus for STATIC.

7 Design Lessons for EBL Systems

The previous section positioned STATIC in the space of mechanisms for optimizing programs. This section abstracts away from details of STATIC and identifies general lessons that can be learned from the STATIC case study.

There are many different ways to prove that a training example is subsumed by a target concept. Each proof gives rise to a different sufficient condition for the concept, and some of the conditions are considerably more general than others. Since EBL merely computes the weakest preconditions associated with a *particular* proof, it is by no means guaranteed to find the most general, or logically weakest, sufficient condition. As a result, EBL frequently derives overly-specific control knowledge [16].

A solution to this problem, embodied by both PRODIGY/EBL and STATIC, is to increase the range of proofs, or analyses, available to the learning module by providing it with a meta-level theory containing multiple target concepts such as failure, success and goal interaction [29]. Increasing the range of proofs available to the learning module highlights the problem of choosing an appropriate analysis of the problem solver's behavior. PRODIGY/EBL makes this choice by generating a large number of control rules and heuristically evaluating their "utility" on a set of training problems. In contrast, STATIC performs no *post hoc* utility evaluation. Instead, STATIC embodies a number of design principles that enable it to generate and select superior proofs *a priori*.

These principles suggest general lessons for the design of EBL systems:

- Training examples can pinpoint learning opportunities, but should not determine EBL's explanations. In contrast to existing EBL systems, *what* EBL explains and *how* it is explained should be determined separately [44].

- A partial meta-level theory encoding sufficient (or necessary) conditions for EBL's target concepts is often preferable to a stronger theory, specifying necessary and sufficient conditions for the concepts, but resulting in more complex explanations [16].

- Nonrecursive explanations are often preferable to recursive ones [15].

- Typically, EBL's proofs correspond to AND trees but, often, a case analysis involving several examples (corresponding to an AND/OR tree), is required to identify the most general sufficient condition for the target concept [16].

- The Problem Space Graph (PSG) is a compact representation that facilitates the implementation of the above ideas. In particular, the PSG facilitates the kind of logical simplification necessary to increase the generality and reduce the match cost of EBL-style control knowledge.

Each principle is illustrated by specific examples below.

7.1 The Pitfalls of Training Examples

Most existing EBL systems generate explanations by directly translating a trace of the problem solver's behavior, when solving a training problem, into a logical proof [27, 33, 35, 47, 52, 56]. As a result, the problem solver's behavior on a training problem determines not only *what* EBL will explain but also *how* the explanation is constructed, yielding overly-specific control knowledge in many cases. The fundamental point of this section is that these decisions should be made independently.

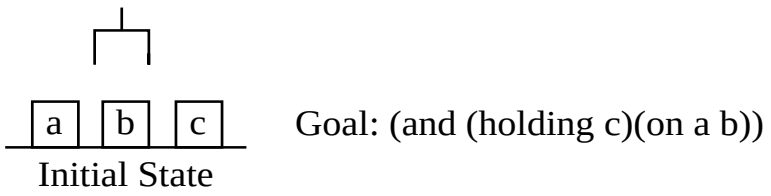

Figure 3: A Blocksworld problem that results in goal clobbering.

Consider the following Blocksworld example. When presented with the training problem in Figure 3, PRODIGY/EBL learns the overly-specific rule in Table 20. PRODIGY/EBL explains how achieving the `holding` subgoal, followed by the `on` subgoal, results in a goal interaction. PRODIGY/EBL's explanation is based on the *particular* plan PRODIGY used in this training problem. PRODIGY/EBL shows that the plan found by PRODIGY results in a goal interaction. To prove that the interaction is necessary (in the sense of Section 4), PRODIGY/EBL analyzes other attempts to achieve `(holding c)`, *given the block configuration in its training problem.* PRODIGY/EBL is able to prove that all other attempts to achieve `(holding c)` fail, and the interaction observed in the training problem is necessary.

```
(EBL-ON-BEFORE-HOLDING
 (if
 (and (current-node Node)
          (candidate-goal Node (holding Block-X))
          (candidate-goal Node (on Block-Y Block-Z))
***    (known Node (on-table Block-X))
***    (known Node (on-table Block-Y))
 (then (prefer goal (on Block-Y Block-Z) (holding Block-X))))
```

Table 20: A (simplified version of) the control rule learned by PRODIGY/EBL based on the problem in Table 3. The ''***'' denote extraneous conditions that make the rule overly-specific and increase its match cost.

Unfortunately, the second part of the explanation refers to the fact that blocks `a` and `c` are on the table. Since these features of the training problem are mentioned in PRODIGY/EBL's

explanation, they are "mechanically" incorporated into the antecedent of PRODIGY/EBL's control rule (Table 20). Yet, the location of the two blocks is incidental to the goal interaction that occurred. The goal interaction occurs *regardless* of the blocks' location. Since PRODIGY/EBL is focused on the particular block configuration encountered in its training problem, it overlooks this fact and includes two extraneous conditions in the antecedent of its control rule.

```
(STATIC-ON-BEFORE-HOLDING
 (if (and (current-node Node)
          (candidate-goal Node (holding Block-X))
          (candidate-goal Node (on Block-Y Block-Z))))
 (then (prefer goal (on Block-Y Block-Z) (holding Block-X))))
```

Table 21: STATIC's corresponding control rule.

In contrast to PRODIGY/EBL, STATIC discovers that *all* plans for achieving `on` clobber the `holding` subgoal. As described in Section 4, STATIC computes (a subset of) the necessary effects of achieving the `on` subgoal, and establishes that one of the necessary effects is `(arm-empty)`, which clobbers the `(holding c)` subgoal. Relying on this analysis, STATIC is able to acquire the more general rule in Table 21. STATIC's analysis is more global than PRODIGY/EBL's because STATIC (implicitly) considers *all* plans for achieving `on` whereas PRODIGY/EBL analyzes only the plans that would succeed in PRODIGY/EBL's training problem, conjoining the conditions under which other plans fail to the antecedent of its control rule.

The above example demonstrates the benefits of a looser coupling between explanations and training examples (cf. [31]). Whereas PRODIGY/EBL's explanations are *determined* by its training examples, STATIC's explanations are based on a global analysis of the partially evaluated problem-space definition, which often yields better control rules. The DYNAMIC system, which is based on this design principle, utilizes both training problems and problem-space analysis to generate PRODIGY control rules [44]. Like PRODIGY/EBL, DYNAMIC utilizes training problems to pinpoint learning opportunities but, like STATIC, DYNAMIC generates control rules via PSG analysis. Thus, DYNAMIC combines the focus and distribution-sensitivity of EBL with the global analysis performed by STATIC. Given the training problem in Figure 3, DYNAMIC is able to generate the maximally general control rule in Table 21.

7.2 Meta-Level Theory Design

Whether training examples are used or not, the generality of acquired control knowledge depends on the precise encoding of EBL's theory.[10] The theory used by PRODIGY/EBL to

[10] This section is based on joint work with Steve Minton, which is described in [16].

construct explanations has two components: the domain theory and the meta-level theory. The domain theory consists of the operators and inference rules provided by the user. The meta-level theory consists of the detailed definitions of the target concepts (e.g., success, failure, and goal-interference) specified by "EBL engineers." Since the point of automatically learning control knowledge is to relieve the user from having to consider control issues when specifying a domain, it is unreasonable to require a naive user to write domain operators so that PRODIGY/EBL will learn effective control rules. The meta-level theory, on the other hand, is a domain-independent theory, crafted by an "EBL engineer." It is not used by the problem solver, and does not change from problem space to problem space. Thus, the cost of designing the theory can be amortized over the life-time of the learning module. For this reason, it *is* reasonable to engineer the meta-level theory in order to improve EBL's performance.

Although it may sound obvious, engineering the meta-level theory so that the "right" explanations are produced is one of the least-appreciated, but most critical, aspects of building an EBL system [32, 31, 28]. The following informal design principle can be used to guide this process: design a theory that "proves as little as possible." The more compact the proof, the less likely it is that irrelevant conditions will "creep" into its weakest preconditions. This principle has two corollaries. Instead of encoding the full necessary *and* sufficient conditions for each target concept of interest, develop partial concept specifications that:

1. Encode simple sufficient (but not necessary) conditions.
   For example, STATIC's theory of failure, defined implicitly by the algorithm in Table 8, merely analyzes failures due to goal-cycles and unachievable preconditions, and ignores more subtle failures that are analyzed by PRODIGY/EBL (e.g. state-loops). Likewise, as pointed out in Section 4, STATIC only analyzes binary goal interactions, ignoring more complex N-ary goal interactions. In both cases STATIC's analysis does not cover every possible instance of its target concepts. Instead, STATIC focuses on sufficient conditions for its concepts that yield compact proofs. Instances of this idea appear in PRODIGY/EBL as well.

2. Encode simple necessary (but not sufficient) conditions.
   Consider, for example, the theory PRODIGY/EBL uses to learn from state-loops. A state-loop occurs when the problem solver finds itself in a state that previously occurred. This is considered a failure point, since if a solution exists, there must exist a path to the solution that does not involve a state-loop. Thus, PRODIGY backtracks whenever it encounters a state-loop. PRODIGY/EBL learns to avoid state-loops by proving they will necessarily occur, given a certain control choice, computing the weakest preconditions of that proof, and avoiding that choice when the weakest preconditions are matched in future problem solving. Unfortunately, due to the complexity of this proof, PRODIGY/EBL often generates overly-specific control rules when analyzing state-loops.

   STATIC's analysis is much simpler. Since every operator application changes PRODIGY's state, state-loops can only occur as a result of goal clobbering. Thus, goal clobbering is

a *necessary condition* for state-loops. STATIC ignores state-loops and merely attempts to avoid goal clobbering. When it succeeds, state-loops are avoided "for free." Thus, instead of analyzing state-loops, STATIC analyzes a necessary condition for state-loops, which yields superior control knowledge. See Appendix A for a striking illustration of the effectiveness of this approach. In general, avoiding a necessary condition for a phenomenon suffices to eliminate the phenomenon altogether.

Encoding the meta-level theory in this manner trades completeness for effectiveness. STATIC seeks to generate effective control knowledge in frequently occurring, easily-handled cases, while disregarding others. Future work may reveal methods for deriving compact proofs based on complete encodings of various target concepts but, in the meantime, STATIC prefers not to "bite off more of the target concept than it can chew."

7.3 Nonrecursive Explanations

Once a meta-level theory is in place, a learning system must decide which target concept to learn from in analyzing a particular training example (or a particular PSG node). STATIC automatically enforces the following criterion: *learn only from the target concepts that yield nonrecursive proofs.*[11] This design choice is motivated, in part, by the structural theory of EBL which shows that learning from recursive explanations is problematic due to the recursion-depth-specificity and worst-case exponential match cost of "recursive control rules" [15] and, in part, by the difficulty of deciding when to terminate the unfolding of recursion during partial evaluation (Section 2.3).

In the presence of multiple target concepts such as success and failure, STATIC's selective learning criterion turns out to be particularly effective. In the Blocksworld, for example, clearing a block is a recursive operation. Clearing the bottom block of an N-block tower requires N-1 calls to the UNSTACK operator. As a result, when learning from success, PRODIGY/EBL forms distinct rules for the two-block tower, the three-block tower, and so on. Each rule is specific to towers of a given height. This is an instance of the infamous generalization-to-N problem [6, 45, 53]. As the state size increases, and increasingly taller towers are possible, more rules are necessary for guiding PRODIGY to choose UNSTACK. STATIC avoids the generalization-to-N problem by generating control rules based on nonrecursive proofs. In fact, utilizing five nonrecursive failure proofs, STATIC is able to generate all the control rules necessary to make appropriate operator and bindings choices in Blocksworld problems of arbitrary size. The nonrecursive proofs yield rules that are compact, general, and cheap-to-match.

Nonrecursive proofs are not *guaranteed* to yield effective rules. Motivated by STATIC, Gratch and DeJong [19] modified PRODIGY/EBL to learn only nonrecursive rules and to omit utility evaluation. This variant of PRODIGY/EBL (called NONREC) performed quite poorly in their experiments, demonstrating that it is quite possible for an EBL system to learn an

[11] See [15] for a precise definition of the term "nonrecursive proof."

ineffective set of nonrecursive rules.[12] Gratch and DeJong's experiment demonstrates the value of the other design ideas, beyond learning from nonrecursive explanations, which are incorporated into STATIC.

7.4 Logical Simplification

One important advantage STATIC has over NONREC, and over EBL systems is general, is its ability to use its PSG representation to facilitate the logical simplification of the control knowledge it generates. EBL can be regarded as incrementally converting the target-concept definition into a DNF expression where each disjunct is a sufficient condition learned from a training example. "Pure" EBL, as described in [11, 36], does not attempt to logically simplify the sufficient conditions it generates, a process that typically requires combining multiple sufficient conditions or disjuncts. For this reason, pure EBL frequently derives overly-specific control knowledge [15, 28, 29].

Consider a simple schematic example taken from [16]. Suppose EBL learns two conditions of the form:
`C if` $(P \wedge Q)$
`C if` $(P \wedge \bar{Q})$.[13]
That is, there is a proof of the target concept whose weakest preconditions are $(P \wedge Q)$, and a proof whose weakest preconditions are $(P \wedge \bar{Q})$). Since either Q or $\bar{Q}$ is always true, a more general condition holds. Namely:
`C if` P
However, pure EBL will not identify this condition.

To address this and related problems, EBL systems such as PRODIGY/EBL include heuristic simplification methods that attempt to improve the generality of EBL's conditions via truth-preserving transformations [29, 47]. Unfortunately, these methods can only work if EBL encounters the appropriate training problems. In the above example, if EBL never learns the second sufficient condition, it is "stuck" with an overly-specific condition. Often, a case analysis involving several examples is required to identify the most general sufficient condition for the target concept, and there is no guarantee that EBL will encounter these examples.

To illustrate this point, consider the schematic PSG shown in Figure 4. Suppose EBL is trying to prove that the goal g fails, and it encounters a training problem in which operator O_1 fails because its precondition P_1 is not true, and O_2 fails because P_2 is not true. EBL will then learn that g fails if $P_1 \wedge P_2$. In contrast, STATIC computes the full (nonrecursive) conditions under which the operators fail. By propagating this information up the PSG, and simplifying the failure condition generated at each node, STATIC is able to perform

[12] It interesting to note, though, that when NONREC was allowed to carry out utility evaluation it actually outperformed PRODIGY/EBL in two of the three problem spaces in which the systems were compared. Since PRODIGY/EBL (and consequently NONREC) generates large numbers of rules, relying on its utility evaluation module to prune ineffective ones, it may be argued that this is a more appropriate comparison.

[13] $\bar{Q}$ denotes the negation of Q.

simplifications that EBL may miss. For instance, STATIC realizes that operator O_2 will fail even if P_2 is true. Thus, $\bar{P}_1$ is sufficient to prove that g fails.

Since STATIC performs a more complete problem-space analysis, that is independent of training examples, it has an advantage over EBL. Whereas EBL's proof trees are typically AND trees, the proof trees that facilitate logical simplification are AND/OR trees of the sort analyzed by STATIC in considering multiple backward-chaining paths through the PSG. Thus, STATIC has an advantage in performing the logical simplifications that increase the generality of control rules and, typically, reduce their match cost. Moreover, the PSG representation preserves the tree structure of the problem space, facilitating *local* simplifications. This structure is lost when EBL attempts to simplify the DNF expression it is forming by "compressing" multiple disjuncts.

STATIC: fails(g) if $\bar{P}_1$.
EBL: fails(g) if $\bar{P}_2$ and $\bar{P}_1$.

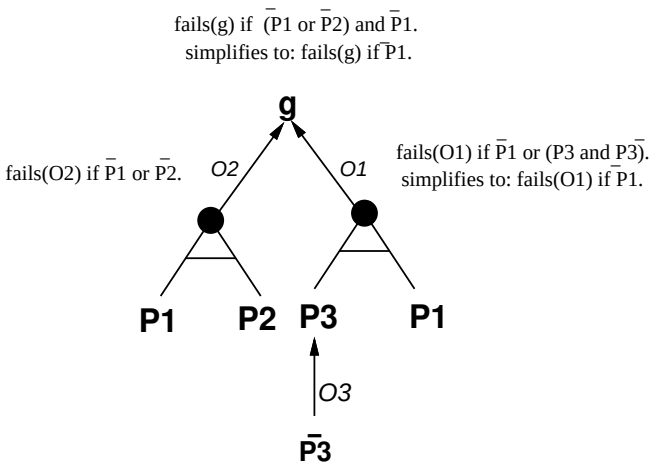

Figure 4: A Schematic PSG illustrating how STATIC is able to derive more general control rules than EBL by performing logical simplification as it traverses the PSG.

7.5 The Compactness of PSGs

Beyond preserving the AND/OR structure of the problem space, which facilitates logical simplification, the tree-structured PSG representation is often *substantially smaller* than the full DNF expansion of a target-concept definition for concepts such as success or failure.[14] Consider a simple example where n operators can potentially achieve some goal g, and each operator has p unachievable preconditions. In this case, the full DNF expansion of

[14]Furthermore, unlike standard EBL systems, static analysis of PSGs obviates the expensive process of repeatedly invoking the problem solver to solve training problems.

the failure target concept contains p^n terms, one term for each distinct combination of unsatisfied preconditions under which the goal g fails. The PSG, in contrast, is exponentially smaller containing only $O(np)$ nodes. Suppose further that the operators all share a common precondition. STATIC will detect this at the PSG node for g, and note that when this precondition cannot be achieved, g cannot be achieved. The number of training examples required for EBL, aided by simplification, to reach the same conclusion is $O(p^n)$ in the worst case.

8 Conclusion

The introduction posed two questions: can static analysis produce effective control knowledge? And can it do so in reasonable time? The STATIC case study has answered both questions affirmatively. When tested in each of PRODIGY/EBL's benchmark problem spaces, STATIC generated search-control knowledge that was up to three times as effective as PRODIGY/EBL's, and did so from twenty-six to seventy-seven times faster. As with any case study, these experimental results are existentially quantified: a PE-based static analyzer, such as STATIC, will efficiently generate effective control knowledge in *some* but not all problem spaces, for *some* but not all problem solvers. With this caveat in place, the results support the structural thesis, formulated and argued for in [15]. The thesis states that *there is a correspondence between the PSG representation and the proofs used by EBL systems to generate search-control knowledge. As a result, automatic PSG analysis can be used to compute the weakest preconditions of the proofs and to discriminate between recursive and nonrecursive proofs.* Furthermore, the STATIC case study demonstrates the effectiveness of PSG analysis as a method of generating search-control knowledge for general problem solvers.

Finally, note that PSG analysis need not be completely static. As DYNAMIC [44] demonstrates, training problems can be used to focus the analysis. Indeed, DYNAMIC is just one example of the gamut of hybrid PE/EBL systems. Such systems can tailor their analysis to the problem distribution encountered, but still benefit from the PSG representation and associated analysis techniques.

Acknowledgments

This paper is based, largely, on my dissertation work at Carnegie Mellon University, which was supported by an AT&T Bell Labs Ph.D. Scholarship. More recent work on STATIC and DYNAMIC was supported, in part, by National Science Foundation Grant IRI-9211045 and by Office of Naval Research Grant 92-J-1946. Thanks are due to my dissertation advisor, Tom Mitchell, to the members of my committee, Jaime Carbonell, Paul Rosenbloom, and Kurt Vanlehn, and to Allen Newell for their impact on the dissertation. Special thanks go to Steve Minton—whose own dissertation made studying PRODIGY/EBL possible. Joe Bates, Craig Knoblock and Steve Minton contributed some of the low-level routines used by STATIC. The anonymous reviewers, Mike Barley, Haym Hirsh, Yan-Bin Jia, Craig Knoblock,

Alicia Pérez, Armand Prieditis, Julie Roomy, Manuela Veloso, and, especially, Frank van Harmelen provided helpful comments on earlier drafts. Alicia Pérez, Julie Roomy, and Rob Spiger designed and implemented extensions to STATIC, helping to refine the description of the algorithm presented here.

A Sample Control Rules

This appendix exhibits a striking example of the differences which can be found between PRODIGY/EBL's rules and STATIC's rules.[15] PRODIGY/EBL learned the following rule by analyzing a state-loop and retained it even after utility evaluation. STATIC learned the corresponding rule by analyzing necessary goal clobbering.

```
(EBL-REJECT-HOLDING
 (if (and (current-node R169)
          (candidate-goal R169 (holding R167))
            (known R169 (on-table R167))
            (is-top-level-goal R169 (on R166 R165))
            (known R169 (on-table R166))
            (not-equal R166 R167)
            (forall (R164)
                    (known R169 (object R164))
                     (or (and (not-equal R164 R167)
                              (known R169 (not (holding R164))))
                         (and (is-equal R167 R164)
                              (forall (R163)
                                  (known R169 (object R163))
                                  (or (is-equal R163 R164)
                                      (and (known R169 (on-table R163))
<<< start back on the left <<<<<
(or (and (is-equal R163 R165)
         (forall (R162)
                 (known R169 (object R162))
                 (or (and (not-equal R162 R166)
                          (or (known R169 (not (holding R162)))
                              (is-equal R162 R164)))
                     (and (is-equal R166 R162)
                          (forall (R161)
                                  (known R169 (object R161))
                                  (or (is-equal R161 R162)
                                      (is-equal R161 R164)
                                      (is-equal R161 R165)))))))
    (is-equal R163 R166)))))))))
(then (reject goal (holding R167))))
```

[15] In many cases, PRODIGY/EBL and STATIC generate similar if not identical rules.

STATIC's corresponding rule, learned by analyzing goal clobbering:

```
(STATIC-PREFER-ON
 (if (and (current-node V95)
          (candidate-goal V95 (on V2 V3))
          (candidate-goal V95 (holding V49))))
 (then (prefer goal (on V2 V3) (holding V49))))
```

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles Techniques and Tools*. Adison Wesley, 1987.

[2] N. Bhatnagar and J. Mostow. Adaptive search by explanation-based learning of heuristic censors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[3] J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Pérez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University, 1992.

[4] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–378, 1987.

[5] J. Cheng and K. B. Irani. Ordering problem subgoals. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 931–936, Detroit, Michigan, 1989. Morgan Kaufmann.

[6] P. Cheng and J. G. Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 490–495, 1986.

[7] J. Christensen. A hierarchical planner that creates its own hieararchies. In *Proceedings of Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[8] A. P. Danyluk. The use of explanations for similarity-based learning. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, Aug. 1987.

[9] J. Darlington. An experimental program transformation and synthesis system. *AI*, 16(1):1–46, March 1981.

[10] C. Dawson and L. Siklossy. The role of preprocessing in problem-solving systems. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

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

[12] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[13] O. Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, Carnegie Mellon University, 1990. Available as technical report CMU-CS-90-185.

[14] O. Etzioni. Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[15] O. Etzioni. A structural theory of explanation-based learning. *Artificial Intelligence*, 1993. To appear. Also available as University of Washington technical report 91-09-02.

[16] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In *Proceedings of the Ninth International Conference on Machine Learning.* Morgan Kaufmann, July 1992.

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

[18] J. Gratch and G. Dejong. A hybrid approach to guaranteed effective control strategies. In *Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann, 1991.

[19] J. Gratch and G. Dejong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of AAAI-92.* AAAI Press, 1992.

[20] H. Hirsh. Combining empirical and analytical learning with version spaces. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 29–33, Ithaca, New York, 1989.

[21] S. Kedar-Cabelli and L. T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, 1987.

[22] S. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

[23] C. Knoblock, S. Minton, and O. Etzioni. Integrating abstraction and explanation-based learning in Prodigy. In *Proceedings of the Ninth National Conference on Artificial Intelligence.*, 1991.

[24] C. A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Available as technical report CMU-CS-91-120.

[25] H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of PROLOG. In *Proceedings of The Ninth Conference on the Principles of Programming Languages (POPL)*, pages 255–267, Albuquerque, NM, 1982.

[26] R. A. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22:572–595, 1974.

[27] J. E. Laird, P. S. Rosenbloom, and N. A. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[28] S. Letovsky. Operationality criteria for recursive predicates. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[29] S. Minton. *Learning Effective Search Control Knolwedge: An Explanation-Based Approach*. PhD thesis, Carnegie Mellon University, 1988. Available as technical report CMU-CS-88-133.

[30] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*. Morgan Kaufmann, 1988.

[31] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3), March 1990.

[32] S. Minton, J. G. Carbonell, O. Etzioni, C. A. Knoblock, and D. R. Kuokka. Acquiring effective search control rules: Explanation-based learning in the Prodigy system. In *Proceedings of the Fourth International Workshop on Machine Learning*, 1987.

[33] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as technical report CMU-CS-89-103.

[34] S. Minton, C. A. Knoblock, D. R. Kuokka, Y. Gil, R. L. Joseph, and J. G. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Carnegie Mellon University, 1989.

[35] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J. C. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, 1991.

[36] T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[37] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In S. Michalski, Ryszard, G. Carbonell, Jaime,

and T. M. Mitchell, editors, *Machine Learning An Artificial Intelligence Approach (volume I)*. Morgan Kaufmann, 1983.

[38] S. S. Muchnick and N. D. Jones. *Program Flow Analysis Theory and Applications*. Prentice Hall, 1981.

[39] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice Hall, 1972.

[40] D. Ourston and R. J. Mooney. Changing the rules: a comprehensive approach to theroy refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence.*, 1990.

[41] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3), September 1983.

[42] M. Pazzani. *Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods*. PhD thesis, University of California, Los Angeles, 1988.

[43] J. Penberthy and D. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *To appear in Proceedings of KR-92*, October 1992.

[44] M. A. Pérez and O. Etzioni. DYNAMIC: a new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning*. Morgan Kaufmann, July 1992. An expanded version available as technical report CMU-CS-92-124.

[45] A. E. Prieditis. Discovery of algorithms from weak methods. In *Proceedings of the international meeting on advances in learning*, pages 37–52, 1986.

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

[47] A. E. Prieditis and J. Mostow. Prolearn: towards a Prolog interpreter that learns. In *Proceedings of the National Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[48] S. Rajamoney and G. DeJong. the classification, detection and handling of imperfect theory problems. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, Aug. 1987.

[49] K. R. Ryu and K. B. Irani. Learning from goal interactions in planning: goal stack analysis and generalization. In *Proceedings of AAAI-92*. AAAI Press, 1992.

[50] M. J. Schoppers. *Representation and Automatic Synthesis of Reaction Plans*. PhD thesis, University of Illinois at Urbana-Champaign, 1989. Available as technical report UIUCDCS-R-89-1546.

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

[52] J. W. Shavlik. Acquiring recursive concepts and iterative concepts with explanation-based learning. *Machine Learning*, 5(1), 1990.

[53] J. W. Shavlik and G. F. DeJong. Building a computer model of classical mechanics. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pages 351–355, 1985.

[54] D. Smith and M. Genesereth. Ordering conjunctive queries. *artificial intelligence*, 26(2):171–215, May 1985. Also Stanford HPP Report HP-82-9.

[55] D. E. Smith. *Controlling Inference*. PhD thesis, Stanford, 1986. Available as technical report STAN-CS-86-1107.

[56] D. Subramanian and R. Feldman. The utility of EBL in recursive domain theories. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[57] J. D. Ullman. *Database and Knowledge-base systems*, volume II. Computer Science Press, 1989.

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

[59] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36, 1988. Research Note.

[60] D. Warren. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the Seventh VLDB conference*, pages 272–281, 1981. IEEE.