Appears in *SIGART Bulletin,* Volume 5, Number 1, pp. 12-21, Jan, 1994

Integrating ILP and EBL

Raymond J. Mooney and John M. Zelle
Department of Computer Sciences
University of Texas
Austin, TX 78712
mooney@cs.utexas.edu, zelle@cs.utexas.edu

Abstract

This paper presents a review of recent work that integrates methods from Inductive Logic Programming (ILP) and Explanation-Based Learning (EBL). ILP and EBL methods have complementary strengths and weaknesses and a number of recent projects have effectively combined them into systems with better performance than either of the individual approaches. In particular, integrated systems have been developed for guiding induction with prior knowledge (ML-Smart, FOCL, GRENDEL) refining imperfect domain theories (FORTE, AUDREY, Rx), and learning effective search-control knowledge (AxA-EBL, DOLPHIN).

1 Introduction

Inductive Logic Programming (ILP) and Explanation-Based Learning (EBL) are two subtopics of Machine Learning that have some important commonalities despite their different histories and methods. ILP [31] focuses on induction of logic programs from examples and has roots in early work in generalization in logic [35] and logic-program synthesis [44]. EBL [25, 10] focuses on improving the efficiency of a problem solver using deductive methods and has roots in early work in learning in planning [13], natural language understanding [8], and mathematical problem solving [23].

Despite these differences, the approaches also have two important similarities. First, both ILP and EBL methods employ background knowledge in learning. ILP methods generalize an existing domain theory by inductively adding new rules that utilize existing concepts in the theory. EBL methods use an existing domain theory to deductively explain examples and use the resulting explanations to focus learning. Second, both ILP and standard EBL methods traditionally represent knowledge in Horn-clause logic, making it easy to apply both methods to the same knowledge base.

A strength of ILP methods is their ability to induce complex concepts represented in an expressive language. A weakness of ILP methods is their inability to use existing knowledge to guide the search through the extremely large space of possible hypotheses. A strength of EBL methods is their ability to use explanations (deductive derivations) to guide the learning of concepts and problem-solving knowledge. A weakness of traditional EBL methods is their inability to handle incomplete or incorrect domain theories. A number of recent projects in machine learning attempt to integrate the two approaches in order to exploit their complementary strengths and compensate for their individual weaknesses.

Attempts to integrate ILP and EBL fall into two broad areas: *concept learning* and *control learning.* ILP/EBL research in concept learning focuses on learning concept definitions from both classified examples and an existing incomplete and/or incorrect domain theory. Two general approaches to integrating ILP and EBL in concept learning have been explored. One approach, *knowledge-guided induction,* uses the existing domain theory to guide the learning of a separate concept definition, e.g. [2, 34, 7]. The other approach, *theory refinement,* uses the examples to modify the existing domain theory in an attempt to improve its accuracy, e.g. [39, 52, 47].

ILP/EBL research in control learning specifically focuses on learning search-control knowledge that improves the performance of an existing logic program or problem solver. Sample problems are used to generate a separate set of control examples describing appropriate and inappropriate contexts for applying operators in the domain theory. A combination of ILP and EBL methods are then used to learn control rules for deciding when to apply the existing operators, e.g. [4, 55]. Control learning is traditionally used to improve the efficiency of a problem solver as a form of *speedup learning* [46]; however, it can also be used to improve accuracy [56].

This paper presents a review of recent research that integrates ILP and EBL methods. We intentionally focus on ILP work that is clearly influenced by ideas from EBL and do not attempt to review other ILP research in the areas of knowledge-based induction and theory refinement, e.g. [11, 44]. The remainder of the paper is organized as follows. Section 2 presents a brief review of traditional EBL for the uninitiated reader. Section 3 reviews ILP/EBL research in concept learning, including work in knowledge-guided induction and theory refinement. Section 4 reviews ILP/EBL research in learning control rules and Section 5 presents our conclusions.

2 Background on EBL

The goal of explanation-based learning is to acquire an efficient concept definition from a single example by using existing background knowledge to explain the example and thereby focus on its important features [9]. The development of EBL was driven by two general goals. First, was the desire to make greater use of background knowledge in learning as opposed to resorting to "tabula rasa" induction. Second, was the desire to employ learning to improve problem solving as opposed to the traditional focus on classification. Historically, EBL methods were developed to improve planning [13], mathematical problem solving [23, 45], and story understanding [8].

The variety of related EBL techniques developed in the early 1980's eventually lead to an effort to unify the various approaches [25, 10]. The resulting definition of the general problem addressed by EBL is shown in Table 1 (from [25]). The task involves using a single example and a domain theory to transform an abstract definition of a concept into an operational definition that is useful for efficient classification. Table 2 shows one of the standard examples of this task, learning a structural definition of a cup from a functional definition, a single example, and a domain theory relating form to function [51, 25]. The basic method for solving the explanation-based generalization problem consists of the fol-

Table 1: The Explanation-Based Generalization Problem.

Given:

• *Goal Concept:* 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 goal concept.

• *Domain Theory:* A set of rules and facts to be used in explaining how the training example is an example of the goal concept.

• *Operationality Criterion:* A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed. Generally, a concept definition that refers only to directly observable properties of the example is assumed to be operational.

Determine:

A generalization of the training example that is a sufficient concept definition for the goal concept and satisfies the operationality criterion.

Table 2: A Sample EBG Problem.

Goal Concept:

```
cup(x) :- stable(x), liftable(x), open_vessel(x).
```

Training Example:

```
owner(obj1,fred).  light(obj1).  color(obj1,red).
partof(h1,obj1).  handle(h1).  bottom(b1).
partof(b1,obj1).  flat(b1).  concavity(c1).
partof(c1,obj1).  up_pointing(c1).
```

Domain Theory:

```
stable(X) :- partof(Y,X), bottom(Y), flat(Y).
liftable(X) :- graspable(X), light(X).
graspable(X) :- partof(Y,X), handle(Y).
open_vessel(X) :- partof(Y,X), concavity(Y),
                  up_pointing(Y).
```

Operationality Criterion: Concept definition must be expressed in terms of structural features used to describe examples (e.g. `light`, `handle`, `flat`, etc.).

lowing two steps:

1. Explain: Construct an explanation using the domain theory that proves that the training example satisfies the definition of the goal concept.

2. Generalize: Determine a set of sufficient conditions under which the explanation structure holds, stated in terms of the operationality criterion.

Standard Prolog deduction is generally used to construct explanations. Several algorithms have been developed for correctly performing the generalization step [28, 17]. These procedures use unification to properly variablize the explanation and thereby generalize the proof as far as possible while maintaining it's correctness. For the example in Table 2, the proof and the generalization are straight-forward. The generalized proof for this example is shown in Figure 1. An operational definition can be obtained by compiling the generalized proof into a new rule. The root of the proof tree forms the consequent of the new rule and the leaves form the antecedents. Below is the compiled rule or *macro rule* for the cup example:

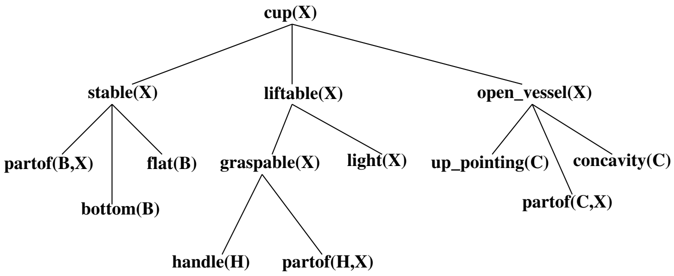

Figure 1: Generalized Explanation for the Cup Example.

```
cup(X) :- partof(B,X), bottom(B), flat(B),
partof(H,X), handle(H), light(X),
partof(C,X), concavity(C), partof(C,X),
up_pointing(C).
```

Unlike the original definition, the learned definition satisfies the operationality criterion since it only refers to observable, structural features. Therefore, instead of performing complex inferencing, direct pattern matching can be used to classify future examples as cups. Explanation-based generalization is closely related to partial evaluation as applied to logic programs and can be viewed as a form of example-guided partial evaluation [50]. It is also closely related to *chunking* as applied to production rules in systems like Soar [43].

Adding macro rules can result in dramatic improvements in efficiency although on many tasks the overhead of matching many specific rules can eventually degrade overall performance [21]. The fact that adding compiled rules can have a negative as well as a positive effect on efficiency is generally referred to as the *utility problem.* A number of techniques have been subsequently developed to help insure the utility of EBL, including simplifying and selectively retaining and utilizing learned rules [22, 26, 16]. As discussed in section 4, combining EBL with ILP is another important approach to addressing the utility problem.

Another well-recognized problem with traditional EBL is the requirement that the domain theory be correct and complete [25]. A number of researchers have combined EBL with various forms of induction in order to address this issue. For example, EBL has been combined with decision-tree induction [32] and neural-network learning [49] in order to refine incomplete and/or incorrect propositional domain theories. The following section discusses methods that combine EBL and ILP in order to learn in the context of incomplete and/or incorrect *first-order* domain theories.

3 ILP/EBL in Concept Learning

As discussed above, research on combining ILP and EBL for general concept learning includes work on both knowledge-guided induction and theory refinement. This section briefly reviews research in both of these areas.

3.1 Knowledge-Guided Induction

All ILP systems perform concept learning in the context of some background knowledge. However, many systems are only concerned with the meanings of the background predicates, not with an intensional representation that might be considered a domain theory. In fact, many popular induction algorithms (e.g. GOLEM [30] and FOIL [38]) represent background theories extensionally as a set of ground facts.

A number of recent systems have attempted to improve the performance of inductive concept learning by utilizing the analytic techniques of EBL and intensional domain theories to provide a bias for induction. We call such an approach knowledge-guided induction.

3.1.1 Theory specialization

One of the earliest uses of EBL techniques to guide concept induction in structured domains was through *theory specialization* [14, 6]. These systems utilize background knowledge represented as an overly-general domain theory. That is, the background theory defines a concept that is a superset of the concept to be learned. More specifically, the theory specialization problem may be formulated as follows:

Given: 1) A domain theory, T_0, defining a superset of the target concept, C
2) Positive and negative examples of C
Find: T_s, a specialization of T_0 defining C.

A concrete example of a theory specialization system to acquire Horn-clause concept definitions is A-EBL (Abductive EBL)[6]. A-EBL uses T_0 to construct all possible proofs of the positive examples. Explanation generalization techniques are then applied to construct operational macros, which serve as a pool of candidate rules for the target theory. A greedy set covering then extracts a small set of rules that are consistent (cover no negative examples) and together cover all of the positive examples.

A-EBL has been tested in a number of domains. In one experiment A-EBL was used to learn concepts in the card game, bridge. Starting with a straight-forward encoding of conventional bidding rules from a bridge text, A-EBL was able to specialize these overly-general guidelines to learn concepts such as "correct opening bid" and "hand of opening strength."

Although theory specialization might seem a very restricted approach for using knowledge to guide induction, it turns out to be surprisingly general. Cohen [5] shows that ANA-EBL, a variant of A-EBL that allows macros to retain some non-operational conditions, is able to exploit many different kinds of background knowledge expressed as overly-general theories. This flexibility derives from viewing the background theory as a specification for the inductive hypothesis space rather than as a concept definition *per se*. This shift in viewpoint underlies the explicit grammatical bias used in GRENDEL, which is discussed below.

3.1.2 ML-SMART

ML-SMART [2] illustrates another approach to integrating EBL and inductive techniques to construct operational concept definitions. Whereas theory specialization systems can only deal with overly-general background theories, ML-SMART is able to handle both overly-general and overly-specific theories.

The backbone of ML-SMART is a (basically) top-down search through the so-called *specialization tree* of the background theory. Each node in this tree contains a (possibly empty) conjunction of operational literals forming a classification rule for the node and a (possible empty) conjunction of non-operational (intermediate) literals which may be operationalized to generate operational literals to further specialize the rule. The root node is the most general literal of the concept being learned.

ML-SMART employs three search operators. A theory-guided component allows a node to be specialized by back-chaining on one of its non-operational literals using a clause from the domain theory. Specialization is also allowed by adding a single arbitrary operational literal. Finally, when a node in the tree is deemed to be overly specific, it may be generalized by dropping an operational literal. A number of statistical, domain-independent, and domain-dependent heuristics are used to guide a best-first search through the specialization tree to find a subset of nodes whose rules cover the positive examples of the concept and exclude the negatives.

A variant of ML-SMART was applied to learning an expert system for a real-world electromechanical fault diagnosis problem [1]. This experiment demonstrated that the use of such techniques could reduce the development time for expert systems and obtain a level of performance comparable to systems developed with classical knowledge-engineering methods.

3.1.3 FOCL

FOCL (First Order Combined Learner) [34] is similar in spirit to ML-SMART, but employs a hill-climbing search based on FOIL. As such, it is most easily understood as method for biasing FOIL with background knowledge.

FOIL constructs a concept definition using a basic covering strategy. A clause is constructed to cover some subset of the positive examples, the covered examples are removed from consideration, and the process repeats until all positive examples have been covered. In building a clause, FOIL adds antecedents one at a time. At each step, FOIL evaluates all possible literals that can be constructed from variablizations of the operational predicates and selects one that maximizes an information-based gain heuristic.

FOCL extends FOIL by including a theory-guided component. In addition to the single literals examined by FOIL the developing clause may also be extended by a conjunction of literals derived from a background theory through an EBL operationalization component. A conjunction of operational literals is found by back-chaining through rules in the domain theory. Choices as to which of several competing rules should be used to reduce a particular literal are made using the FOIL information-gain metric. In this way, the inductive and analytic components are cleanly integrated with a single unified evaluation metric.

Systematic evaluations of FOCL and FOCL-FRONTIER [33], an extension that allows the learning of definitions containing non-operational literals, have been carried out in a number of domains. In the standard ILP domain of classifying illegal king-rook-king chess configurations [38], FOCL was shown capable of learning accurate definitions in the presence of a wide variety of errors that were artificially introduced in the initial domain knowledge. In another experiment, these systems were shown to significantly outperform pure induction in a real-world task customizing an existing expert system for locating phone-line faults.

3.1.4 GRENDEL

GRENDEL [7] is another FOIL-based system that incorporates domain knowledge. Unlike, the previously discussed systems where background knowledge is provided in the form of a Horn-clause theory that somehow approximates the concept, GRENDEL uses an *antecedent description grammar* (ADG) to explicitly represent the inductive hypothesis space. An ADG may be viewed as a sort of extended context-free grammar that describes the space of all clauses that can be used in the concept definition. GRENDEL employs a hill-climbing search in this space in a manner reminiscent of the operationalization process of FOCL

The advantage of the grammatical framework is that this single technique is general enough to take advantage of many varied forms of background knowledge including partially correct theories, programming "cliches", constraints on how predicates may be used, and theories of related concepts. Using such an approach, it is possible to "code-up" different learning biases for various domains within a single framework having a clear interpretation. For example, it is possible to write ADG's that emulate various systems described here including EBL, FOIL, A-EBL, and FOCL.

A disadvantage of the grammatical approach is that it does not operate directly on existing domain knowledge. Whereas other methods assume the existence of (perhaps incorrect) rule-bases which can still provide some guidance for an inductive learner, GRENDEL requires that the knowledge and biases be translated into an ADG. One solution to this problem is to write systems that automatically convert rule-bases into the ADG formalism in a canonical fashion that captures the constraints of interest.

3.2 Theory Refinement

Several recent projects have focussed on combining ideas from ILP and EBL to modify an incomplete and/or incorrect first-order Horn-clause domain theory to make it consistent with a set of training examples. The standard logical definition of ILP only considers adding clauses to a theory in order to allow it to derive a set of positive examples without deriving a set of negative examples [30]. Theory refinement allows for the possibility that the existing background knowledge is incorrect, and therefore also allows existing clauses to be generalized, specialized, and deleted.

The general theory refinement problem is: Given a domain theory T, a set of positive examples E^+ and a set of negative examples E^-, find a revised theory T' such that $T' \vdash E^+$ and $T' \not\vdash E^-$. In order to preserve the correct aspects of the initial theory, it is desirable to modify it as little as possible. Since the minimal semantic change is to simply memorize the incorrect examples as exceptions to the theory, most systems focus on minimal syntactic change in order to force generalization to novel cases. A notion of the *syntactic distance* between two theories can be defined based on the number of primitive edit operations required to transform one theory into the other [27, 54]. Unfortunately, it is computationally intractable to minimize such measures when revising a theory. Therefore, implemented systems must resort to some form of heuristic search in an attempt to minimize syntactic change to the theory.

Researchers familiar with EBL have generally approached the problem of theory refinement by using failed attempts to explain positive examples and incorrect explanations of negative examples to suggest potential revisions to the domain theory. This section reviews several theory refinement systems that take this approach.

3.2.1 FORTE

FORTE [39, 40] performs a hill-climbing search through a space of specializing and generalizing operators in an attempt to find a minimal revision to a theory that makes it consistent with the training examples. FORTE's revision operators include methods from propositional theory refinement [32], first order induction [38], and inverse resolution [29].

Table 3: FORTE's program debugging results.

Program	# of Programs	Training Set Size	Train Time	% Correct
directed path	4	121	87 secs	100%
insert after	9	35	82 secs	100%
merge sort	10	60	199 secs	100%

First, FORTE attempts to prove all positive and negative examples using the current theory. When a positive example is unprovable, some clause in the theory needs to be generalized. All clauses that were backtracked over during the attempted proof are candidates for generalization. When a negative example is provable, some clause needs to be specialized. All clauses that participated in the successful proof are candidates for specialization.

When an error is detected, FORTE identifies all clauses that are candidates for revision. The core of the system consists of a set of operators that generalize or specialize a clause to correctly classify a set of examples. Based on the error, all relevant operators are applied to each candidate clause. The best revision, as determined by classification accuracy on the complete training set, is implemented. This process iterates until the theory is consistent with the training set or until FORTE is caught in a local maximum, i.e. none of the proposed revisions improve overall accuracy.

FORTE's specialization operators include rule deletion and antecedent addition. Several methods are used to determine appropriate additional antecedents to add to an overly-general clause. One is a hill-climbing method based on FOIL [38]. Another is called *relational pathfinding* [41] and adds a sequence of literals that form a relational path linking all of the arguments of the goal predicate. Since it adds multiple literals at once, relational pathfinding helps overcome local minima problems in FOIL.

FORTE's generalization operators include deleting antecedents and adding rules. Antecedents are chosen for deletion using a greedy algorithm that attempts to maximize the number of additional provable positive examples without causing additional provable negatives. New rules are learned using FOIL and relational pathfinding. FORTE also includes two additional generalization operators (identification and absorption) based on *inverse resolution* [29]. These operators introduce new rules based on repeated patterns of literals found in existing rules.

In one test, FORTE was used to automatically debug Prolog programs written by undergraduates for an assignment in a class on programming languages. Students were asked to submit their programs after they had satisfied themselves on paper that they were correct, but before they tried to run them. The student programs were distributed among three problems: finding a path through a directed graph, inserting an element into a list, and merge-sorting a list. Twenty-three distinctly different incorrect programs were collected, representing a wide variety of bugs ranging from simple typographical errors to complete misunderstandings of recursion. FORTE correctly debugged all 23 programs (see Table 3). FORTE has also been used to debug a version of the decision-tree induction program from Bratko's Prolog text [3], and to revise a qualitative model of a portion of the Reaction Control System of the NASA Space Shuttle.

3.2.2 AUDREY and A3

AUDREY [52], AUDREY II [54], and A3 [53] are a series of first-order theory refinement systems that also integrate ideas from ILP and EBL. These systems share features with both FORTE and the propositional theory refinement system EITHER [32] as well as incorporating several unique features.

Like EITHER and unlike FORTE, AUDREY II employs two separate phases. First the theory is specialized to remove any proofs of negatives, and next it is generalized to prove all of the positives. Specialization is performed by a hill-climbing search that at each point selects the clause who's deletion corrects the most false positives. If deleting this clause does not cause any positive examples to become unprovable, it is deleted. Otherwise a FOIL-like method is used to add additional antecedents until all of the provable negative examples for which this clause is responsible are fixed. The original AUDREY did not include clause specialization and instead relied on generalization to re-cover any uncovered positives resulting from clause deletion.

Next, AUDREY II generalizes the theory to cover all of the positive examples. An unprovable positive is selected at random and an abductive process is used to find a single assumption that will make the example provable. If deleting the assumed literal from the relevant clause does not cause negative examples to become provable, then it is deleted. If deletion causes false positives, the system next attempts to replace the assumed literal with a conjunction of new literals learned using FOIL. If this also fails to improve coverage, FOIL is finally used to learn a new set of clauses. This generalization process repeats until all of the positives are covered. AUDREY's use of abduction is similar to EITHER's; however, EITHER was not restricted to making a single assumption and used greedy covering, selecting at each step the most promising abduction from a potentially large set of possibilities generated from all of the unprovable positives.

A3 is the most recent system in the series and adds the ability to revise theories employing negation as failure, i.e. Prolog's `not` operator. The notion of an assumption is generalized to include negated literals of the form `not(P)`, representing the assumption that `P` is not provable. Such assumptions can represent both specializations and generalizations to the theory as appropriate. At each step, a single correcting assumption is found for each incorrectly classified example and the assumption that corrects the most examples is used to form a revision. Unnegated assumptions are used to form a generalization such as antecedent deletion or clause addition. Negated assumptions are used to form a specialization such as clause deletion or antecedent addition.

Both AUDREY II and A3 have been tested on revising randomly corrupted versions of a domain theory for determining whether or not payment is due on a student loan. With a small number of training examples, the revised theories produced for this domain are significantly more accurate than the operational definitions learned by FOCL. Since theory refinement attempts to preserve the structure and content of the domain theory as much as possible, it can have an advantage over knowledge-guided induction as illustrated in this domain.

3.2.3 Rx and LATEX

Rx [47] and LATEX [48] are two recent first-order theory refinement systems developed at the Tokyo Institute of Technology. Rx initially learns an operational concept definition in a manner very similar to FOCL and then "unoperationalizes" this definition back into a revised theory. The general approach is very similar to the reduce, revise, and retranslate method used in the RTLS propositional theory reviser [15]. LATEX is a simple hill-climbing revision system that uses a method based on minimum description length (MDL) to handle noisy data.

The theory revision process in Rx consists of four steps: 1) Operationalization, 2) Specialization, 3) Rule creation, and 4) Unoperationalization. Operationalization expands the theory into a set of operational clauses, removing literals that do not have sufficient information gain according to the FOIL metric (< 0.25). Specialization uses FOIL to add literals to the resulting operational clauses that cover negative examples. Rule creation uses FOIL to add additional clauses that cover any remaining positive examples. Finally, unoperationalization reconstructs a hierarchical domain theory from the revised operational definition. This is accomplished by revising the definition of each subconcept, starting with the deepest (most operational) subconcepts and working bottom-up until the goal concept is reached. The revised definition of each subconcept is constructed by noticing similarities and differences in the literals that FOIL added to the operational clauses expanded from each of the rules for the subconcept. A problem with this approach is that it relies on FOIL adding the same correct literals to each of the resulting operational clauses in order to reconstruct a reasonable subconcept definition. Rx has successfully revised randomly corrupted versions of a simple chemical theory about buffer solutions. The system produced revised theories that were significantly more accurate than those learned by FOIL and correctly revised subconcepts in the theory for which no direct training examples were provided.

LATEX is a quite different approach to theory refinement that employs a unique application of the minimum description length (MDL) principle [42] to effectively learn from an approximate theory and noisy data. The normal approach to using MDL in inductive learning with noisy data is to attempt to induce a theory that minimizes the sum of the number of bits required to represent the theory plus the number of bits required to encode the classification of the training examples given the theory [37]. This allows one to resolve the trade-off between making a theory more complex in order to account for the remaining misclassified data or accepting the simpler theory and treating the remaining data as noise. In order to use MDL in the context of theory refinement, LATEX uses a slight variation in which one attempts to minimize the sum of the number of bits required to encode the changes to the initial theory plus the number of bits required to encode the classification of the training examples given the revised theory. This approach resolves the trade-off between making additional changes to one's initial theory or simply accepting the remaining misclassified data as noise. LATEX uses a very simple hill-climbing revision algorithm that tries all possible single-literal additions and deletions anywhere in the theory and picks the one that minimizes the description length metric given above. Experiments in the standard ILP domain of determining illegal king-rook-king chess configurations [38] demonstrated that this approach could effectively revise theories even in the presence of data with 10% classification noise.

4 ILP/EBL in Control Learning

As previously mentioned, EBL research is rooted in the use of learning to improve the performance of problem-solving systems. One prominent method is the learning of search-control knowledge. The notion of intelligence as controlled search pervades AI. Learning rules to control search can improve both the efficiency and accuracy of a problem solver by eliminating search along paths that do not lead to correct solutions.

4.1 Background

Early systems such as LEX [24] and SAGE [19] used inductive methods to learn operator-selection rules. Using experience from a set of training problems, these systems collected examples where the application of each operator led to successful problem solutions. Induction over the problem-solving states in which various operators were successful or unsuccessful produced operator-selection rules. The use of such rules when solving new problems allows operators to be chosen according to their potential usefulness, limiting search and enhancing performance.

It was quickly realized that a purely inductive approach to the learning of control knowledge ignored a major source of information: the solution sequence of the problem itself. For example, LEX, which learned search control heuristics for solving problems in integral calculus, might require a number of examples to form an inductive generalization such as:

IF the integral is of form:
$< any - fn > * \int r_1 * x^{r_2} dx$ and $r_2 \neq -1$
THEN apply *operator-3*.

However, utilizing a simple domain theory stating that an operator is useful when it leads to a solution allows the learning of this generalization from a single example using EBL techniques [25]. Such techniques formed the basis of the LEX2 system [23]. Subsequent speedup learning research in planning (e.g. PRODIGY [21], STATIC [12]) and production systems (e.g. Soar [18]) has focussed on these analytic methods.

Arguably, one of the reasons for the ascent of EBL over inductive techniques in control-rule learning was the relative lack of good tools for performing induction in structured domains. The recent emergence of new, efficient ILP algorithms (e.g. FOIL and GOLEM), has led to renewed interest in inductive methods for control-rule learning.

Leckie and Zuckerman's GRASSHOPPER [20] uses a modification of FOIL to learn search-control rules in a PRODIGY framework. GRASSHOPPER was shown to outperform PRODIGY/EBL on several standard planning problems. The inductive process was able to learn rules that were relatively simple and covered multiple examples. These two factors serve to increase the *utility* of the resulting rules by lowering the cost of matching the rules to new problem states and insuring that the learned rules have broad applicability. Thus induction helps tame the *utility problem* of control-learning [21] wherein the acquisition of control rules can sometimes degrade the overall performance of a system.

Of course, the purely inductive approach of GRASSHOPPER does not take advantage of the information provided by analytic methods. It would seem that a combination of EBL and ILP might provide the best alternative. A unifying framework that cleanly integrates these techniques can be achieved by considering the problem of control-rule learning within the context of logic programming.

4.2 Controlling Search in Logic Programs

Although most EBL research in logic programming has generally focussed on learning *macros*, [25, 10, 36], Recent work has shown the utility of learning explicit search-control rules within a logic programming framework [4, 55, 56].

The execution of a logic program can be viewed as a problem solving process with a search strategy based on resolution theorem proving. A program executes by finding a constructive proof of a partially instantiated goal given as input. Prolog provides a particular implementation of logic programming using a very simple control strategy incorporating depth-first search with simple backtracking.

Controlling search in this framework can be viewed as a *clause selection* problem [4]. Clause selection is the process of deciding which of several applicable program clauses should be used to solve a particular subgoal during the course of a proof. If the theorem prover always applies an appropriate clause to reduce the current subgoal, the program executes deterministically (without backtracking) and produces only correct solutions.

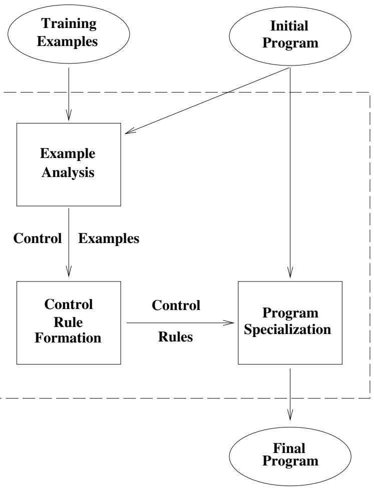

Figure 2: Learning Clause Selection Heuristics

A framework for control-rule learning in Prolog programs is illustrated in Figure 2. The input to the learning system is a Prolog program and a set of training examples, which are fully instantiated examples of the top-level program goal. The output is a modified program that incorporates learned clause-selection heuristics.

When performing speedup learning, the initial program is correct, but potentially inefficient. The set of training examples can be generated by using the initial program to enumerate the solutions to a set of prototypical problems. For programs that compute a function, each solved problem will generate a single (fully instantiated) training example. In the case of program that computes a relation (multiple outputs for a given input) each problem may give rise to multiple training examples.

The three phases of the framework, *example analysis*, *control rule formation*, and *program specialization*, are explained and illustrated by way of a simple example in the following subsections.

4.2.1 Example analysis

During example analysis, the training examples are utilized to identify clauses of the program that give rise to unwanted backtracking or incorrect solutions. Examples of correct and incorrect clause applications are extracted as *control examples* for subsequent control rule induction. A control example for a clause is a (partially) instantiated subgoal to which the clause was applied at some point during the search for a proof

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

permutation([],[]).
permutation([X|Xs],Ys) :-  permutation(Xs,Ys0),
                           insert(X,Ys0,Ys).

insert(X,Xs,[X|Xs]).
insert(X,[Y|Ys0],[Y|Ys]) :-  insert(X,Ys0,Ys).

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

Figure 3: Naive Sorting Program

Table 4: Control Examples for First insert Clause

Positives	Negatives
insert(9,[],A)	insert(9,[5],A)
insert(1,[3,4,5],A)	insert(9,[4,5],A)
insert(5,[],A)	insert(9,[3,4,5],A)
insert(3,[4],A)	insert(9,[1,3,4,5],A)
insert(4,[],A)	insert(5,[4],A)
	insert(5,[3,4],A)

of a training example.

Positive control examples are generated by finding the first proof for each training example using the initial program. The clause applications actually used in the completed proof are retraced, but with the goal having only its input arguments instantiated. For each clause application in the proof, a "snapshot" of the subgoal to which it is applied is saved as a positive control example for that clause. These positive control examples reflect the subgoals to which the clause should be applied when the program is actually being used to generate the expected output from the given input.

The set of positive control examples also implicitly defines a set of negative control examples. Knowing the set of clauses that should be applied to a given subgoal identifies other clauses as those that should *not* be applied. Thus, positive examples for any clause can be considered as negative control examples for any competing clauses.

As an example, consider the naive sorting program in Figure 3, which sorts a list by generating permutations until it finds one that is ordered. Permutations are generated by permuting the tail of the input list and inserting the head somewhere in the permuted tail. The predicate, `insert(Item, List1, List2)`, holds when `List2` is `List1` with `Item` inserted at an arbitrary location. In `naivesort`, `insert` is called with its first and second arguments instantiated to insert items into the permutation, which is returned in the third argument.

In this specification of sorting, computation proceeds "nondeterministically" by generating successive permutations. The nondeterminism of the `permutation` predicate actually arises from the definition of `insert`. Either clause of `insert` may be used to reduce any `insert` subgoal. This nondeterminism could be eliminated by learning a control rule for the first clause that accurately predicts the situations in which an item should be placed at the front of the list.

Given, `naivesort([9,1,5,3,4],[1,3,4,5,9])`, the example analysis phase discovers five examples of correct uses of the clause and six failed attempts. These control examples, shown in Table 4, represent the concept `useful_insert_1`, that is, subgoals to which the first clause of `insert` should be applied. The positive control examples are the subgoals that were solved by the first `insert` clause. The negative control examples for this clause are the subgoals solved by the second clause of `insert`.

```
insert(A, B, [A|B]) :-  useful_insert_1(A, B, [A|B]).
insert(A, [B|C], [B|D]) :- insert(A, C, D).

useful_insert_1(A, [], [A]).
useful_insert_1(A, [B|C], [A,B|C]) :- A =< B.
```

Figure 4: Improved Insert Predicate

4.2.2 Control rule formation

The goal of control-rule formation is to produce a set of control rules specifying the contexts in which it is useful to apply clauses of the initial program. For each program clause, C, a definition of the concept, "subgoals for which C is useful," is needed. Given the positive and negative control examples for a clause, the specific task is to find a set of clauses that cover all (or most) the positive examples, and as few negatives as possible. The algorithm for acquiring control-rules may be purely inductive, purely analytic, or employ a combination of techniques.

Continuing with the naivesort example, a suitable algorithm might produce a control rule such as:

```
useful_insert_1(insert(A,[],[A]))← true
useful_insert_1(insert(A, [B|C],[A,B|C]))← A=<B
```

This simple definition correctly classifies the examples in Table 4. It states that the first clause of `insert` should be used to insert an element into a list that is empty or has a head at least as large as the element being inserted.

4.2.3 Program specialization

In the program specialization phase, the initial Prolog program is modified using the learned control rules so that attempts to use a clause inappropriately fail immediately. This is accomplished by adding the learned control-conditions to the beginning of clauses that give rise to backtracking. In this way, the search space of the Prolog solver is pruned to efficiently produce correct solutions, effectively utilizing the control information without incurring the overhead of a separate interpreter.

Returning to the sorting example, folding the clause selection rules back into the program produces a new definition of `insert` shown in Figure 4. In effect, `permutation` has been modified to produce ordered permutations. Careful inspection shows that this is a version of the insertion sort algorithm, and an $O(n!)$ sort has been "optimized" into an $O(n^2)$ version by learning and incorporating suitable control rules.

4.3 An EBL Approach

One approach to learning search-control in a logic program is EBL-control [5], an application of standard EBL methods to the clause selection problem. Each clause application in a successful proof is "explained" by compiling a macro-rule for the subgoal which extracts the generalized operational conditions that allowed the the clause to successfully solve the subgoal. These macros are then collected as a set of control rules that cover the positive control examples (note that this strategy ignores the negative control examples).

Unfortunately, this approach has some deficiencies. First, it tends to produce control rules containing many (often irrelevant) conditions which make them costly to match against subsequent subgoals and can degrade overall performance rather than improving it. This is a manifestation of the utility problem mentioned above.

A second shortcoming is that this approach only explains the success of a clause application in terms a successful proof of the subgoal to which the clause was applied. This does not guarantee that the proof of the subgoal will be useful in completing the surrounding proof. EBL-control is not able to induce the kind of control rule that would be useful in optimizing the `naivesort` example. Recall that the necessary control rule must decide when to apply the first clause of the definition of `insert`. EBL-control explains this decision by considering the proofs of `insert` subgoals. The proof of the subgoal in this case is just, `true`, because the first clause of `insert` has no antecedents. Hence the only learnable condition is `true` which is, obviously, not a useful heuristic. Put another way, since every application of either clause of `insert` will ultimately succeed, EBL-control will not be able to formulate any control rules for the sorting problem that exclude the negative control examples. This type of problem illustrates the need to consider conditions of the top-level proof that lie outside of the subproof of the immediate goal for which clause selection rules are being learned.

4.4 AxA-EBL

Cohen [4] attacks the utility problem for clause selection by combining EBL with induction to learn a small set of "approximate" control rules with reduced match cost. His algorithm, AxA-EBL (Approximating Abductive EBL), is a variant of the A-EBL concept learning algorithm applied to the problem of learning control rules.

Control-rule formation in AxA-EBL starts with the standard EBL explanations of correct clause applications by compiling out a generalized macro for the subgoal to which the clause was applied. The set of macros for a given clause is expanded into a set of candidate control rules by considering all *k-bounded approximations* of these macros. A k-bounded approximation is formed by dropping all but j conditions from the macro for some $j < k$. AxA-EBL then performs a greedy covering of the positive control examples, selecting control-rules that are consistent (do not cover negative control examples) and maximize the ratio of positives covered to rule (explanation) size.

In a series of experiments utilizing different problem solvers encoded as Prolog programs, AxA-EBL was shown to outperform standard EBL and A-EBL used as a control-rule learner. These domains included a simplified version of the LEX domain employing state-space search with iterative deepening, two standard planning domains (STRIPS-world and Blocks-world) and a bounded depth-first graph search. It is particularly noteworthy that AxA-EBL performed much better than the competing approaches on the graph-search problem which poses severe utility problems for standard EBL.

Unfortunately, AxA-EBL inherits the shortcoming of EBL-control discussed above, which prevents it from learning the desired control rule for `naivesort`. AxA-EBL is also limited by efficiency considerations; the number of candidate rules is exponential in k, and hence only relatively simple approximations may be considered.

4.5 DOLPHIN

DOLPHIN, (Dynamic Optimization of Logic Programs through Heuristics INduction) [55] is an integration of ILP and EBL that extends the AxA-EBL approach. DOLPHIN improves on AxA-EBL in two significant ways. First, like GRASSHOPPER, it employs FOIL, a powerful ILP induction algorithm [38]. Second, DOLPHIN explicitly considers the surrounding proof context during control-rule induction.

DOLPHIN begins by applying EBL techniques to the entire proof-tree of each training example to produce a set of generalized proofs. These generalized proofs of top-level examples can be seen as giving the context for the appropriate application of clauses used within the proof. The operational nodes of the proof represent all of the primitive conditions that had to be satisfied for the proof to succeed. DOLPHIN uses induction to identify a small set of simple tests that provide significant guidance in determining whether the application of a given clause is likely to be part of a complete proof.

Control-rule formation is performed as concept learning over the control examples informed by the generalized proofs of the training examples. DOLPHIN employs the same general covering algorithm as FOIL but modifies the clause construction step. Whereas FOIL considers every possible literal that might be used to extend the current clause, DOLPHIN specializes by considering only operational literals from the generalized proofs of the training problems. With this approach, DOLPHIN is often able to learn exactly the appropriate control-rules for naivesort from a single example. Training with three or more simple examples virtually insures learning of the insertion-sort heuristic.

DOLPHIN has been evaluated in a number of problem-solving domains. Figure 5 shows speedup curves comparing the performance of DOLPHIN to three other approaches in optimizing a Prolog program that implements a means-ends analysis planner for a STRIPS-like robot-world domain. The graph depicts the time required to solve a set of benchmark problems as a function of the number of training examples used to perform program optimization. In addition to EBL-control and AxA-EBL which are the control-rule learning algorithms discussed above, the figure also shows a comparison to a macro-based speedup strategy, EBL-macro.

EBL-macro uses a macro-operator approach reminiscent of the original STRIPS macro-operator learning mechanism [13]. EBL-macro collects a set of top-level macros from the training problems and attempts to solve new problems by direct application of a previously learned macro before resorting to normal planning. As an illustration, for the naivesort example, `naivesort([9,1,5,3,4],X)`, EBL-macro stores the learned rule:

```
naivesort([A,B,C,D,E], [B,D,E,C,A]) :-
     B=<D, D=<E, E=<C, C=<A.
```

Obviously, this is not a useful strategy in the sorting problem as the set of macros essentially memorizes different list orderings and has little ability to generalize to new problems. In planning domains, however, it is often surprisingly successful.

These results are consistent with those in the other test domains showing that DOLPHIN can achieve significant speedup with relatively small sets of training problems.

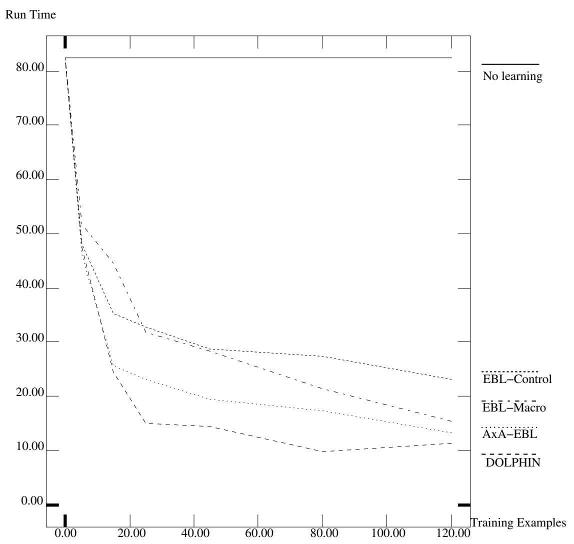

Figure 5: STRIPS-planner Results

5 Conclusion

The research reviewed in this article clearly indicates that ILP and EBL methods can be effectively integrated to produce a system that performs better than either method alone. In particular, ideas from explanation-based learning are useful in guiding induction with prior knowledge, refining incomplete and incorrect domain theories, and improving the learning of search-control knowledge. The fact that several systems that integrate ILP and EBL have already been successfully applied to real-world problems further demonstrates the promise of this approach.

Acknowledgments

The authors' research is supported by the National Science Foundation under grant IRI-9102926 and the Texas Advanced Research Program under grant 003658114.

References

[1] F. Bergadano, A. Giodana, L. Saitta, D. De Marchi, and F. Brancadori. Integrated learning in a real domain. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 322–328, Austin, TX, June 1990.

[2] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 305–317, Ann Arbor, MI, June 1988.

[3] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, Reading:MA, 1990.

[4] W. W. Cohen. Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276, Austin, TX, June 1990.

[5] W. W. Cohen. The generality of over-generality. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 490–495, Evanston, IL, June 1991.

[6] W. W. Cohen. Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, 8(2):167–219, 1992.

[7] W.W. Cohen. Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 102–110, Aberdeen, Scotland, July 1992.

[8] G. DeJong. Generalizations based on explanations. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 67–70, Vancouver, BC, 1981.

[9] G. F. DeJong, editor. *Investigating Explanation-Based Learning*. Kluwer Academic Publishers, Norwell, MA, 1993.

[10] G. F. DeJong and R. J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[11] L. DeRaedt and M. Bruynooghe. An overview of the interactive concept learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–192. Academic Press, New York, 1992.

[12] O. Etzioni. STATIC: A problem-space compiler for PRODIGY. In *Proceedings of National Conference on Artificial Intelligence*, pages 533–540, Anaheim, CA, 1991.

[13] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[14] N. S. Flann and T. G. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4(2):187–226, 1989.

[15] A. Ginsberg. Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777–782, Detroit, MI, July 1990.

[16] J. Gratch and G. DeJong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 235–240, San Jose, CA, July 1992.

[17] S. T. Kedar-Cabelli and L. T. McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the 1987 Machine Learning Workshop*, pages 383–389, Irvine, CA, June 1987.

[18] J. Laird, P. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986.

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

[20] C. Leckie and I. Zukerman. An inductive approach to learning search control rules for planning. In *Proceedings of the Thirteenth International Joint conference on Artificial intelligence*, pages 1100–1105, Chambery, France, 1993.

[21] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, St. Paul, MN, August 1988.

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

[23] T. Mitchell. Learning and problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 1139–1151, Karlsruhe, West Germany, August 1983.

[24] T. Mitchell, T. Utgoff, and R. Banerji. Learning problem solving heuristics by experimentation. In R. Michalski, T. Mitchell, and J. Carbonell, editors, *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Palo Alto, CA, 1983.

[25] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[26] R. J. Mooney. The effect of rule use on the utility of explanation-based learning. In *Proceedings of the Eleventh International Joint conference on Artificial intelligence*, pages 725–730, Detroit, MI, August 1989.

[27] R. J. Mooney. A preliminary PAC analysis of theory revision. In T. Petsche, S. Judd, and S. Hanson, editors, *Computational Learning Theory and Natural Learning Systems, Vol. 3*. MIT Press, Cambridge, MA, in press.

[28] R. J. Mooney and S. W. Bennett. A domain independent explanation-based generalizer. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 551–555, Philadelphia, PA, August 1986.

[29] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, Ann Arbor, MI, June 1988.

[30] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–297. Academic Press, New York, 1992.

[31] S. H. Muggleton, editor. *Inductive Logic Programming*. Academic Press, New York, NY, 1992.

[32] D. Ourston and R. Mooney. Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 815–820, Detroit, MI, July 1990.

[33] M. Pazzani and C. Brunk. Finding accurate frontiers: A knowledge-intensive approach to relational learning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 328–334, Amherst, MA, 1993.

[34] M. Pazzani and D. Kibler. The utility of background knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[35] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence (Vol. 5)*. Elsevier North-Holland, New York, 1970.

[36] A. Prieditis and J. Mostow. Prolearn: Towards a Prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, Jul 1987.

[37] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80:227–248, 1989.

[38] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[39] B. Richards and R. Mooney. First-order theory revision. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 447–451, Evanston, IL, June 1991.

[40] B. L. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, to appear.

[41] B.L. Richards and R. J. Mooney. Learning relations by pathfinding. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, July 1992.

[42] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[43] P. Rosenbloom and J. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, PA, Aug 1986.

[44] E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

[45] B. Silver. Precondition analysis: Learning control information. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An AI Approach, Vol. II*, pages 647–670. Morgan Kaufman, San Mateo, CA, 1986.

[46] P. Tadepalli, editor. *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*, Aberdeen, Scotland, July 1992.

[47] S. Tangkitvanich and M. Shimura. Refining a relational theory with multiple faults in the concept and subconcepts. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 436–444, Aberdeen, Scotland, July 1992.

[48] S. Tangkitvanich and M. Shimura. Learning from an approximate theory and noisy examples. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 466–471, Washington, D.C., July 1993.

[49] G. G. Towell, J. W. Shavlik, and Michiel O. Noordewier. Refinement of approximate domain theories by knowledge-based artificial neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861–866, Boston, MA, July 1990.

[50] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

[51] P. H. Winston, T. O. Binford, B. Katz, and M. Lowry. Learning physical descriptions from functional definitions, examples, and precedents. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 433–439, Washington, D.C., Aug 1983.

[52] J. Wogulis. Revising relational domain theories. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 462–466, Evanston, IL, June 1991.

[53] J. Wogulis and M. Pazzani. Handling negation in first-order theory revision. In *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, pages 36–46, Chambery, France, 1993.

[54] J. Wogulis and M. Pazzani. A methodology for evaluating theory revision systems: Results with Audrey II. In *Proceedings of the Thirteenth International Joint conference on Artificial intelligence*, pages 1128–1134, Chambery, France, 1993.

[55] J. M. Zelle and R. J. Mooney. Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint conference on Artificial intelligence*, pages 1106–1111, Chambery, France, 1993.

[56] J. M. Zelle and R. J. Mooney. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 817–822, Washington, D.C., July 1993.