Constraints for Predicate Invention

Ruediger Wirth
wirth@ics.uci.edu
Paul O'Rorke
ororke@ics.uci.edu

Technical Report 91–23

February 28, 1991
Revised July 17, 1991

This is a revised and expanded version of Wirth, R., & O'Rorke, P. (1991), Constraints on predicate invention. In L. A. Birnbaum & G. C. Collins (Eds.), *The Eighth International Workshop on Machine Learning* (pp. 457–461). Evanston, IL: Morgan Kaufmann. Research supported in part by National Science Foundation Grant Number IRI-8813048, Douglas Aircraft Company, and the University of California Microelectronics Innovation and Computer Research Opportunities Program.

We thank Mike Pazzani, Dennis Kibler and the graduate students of the AI & ML community at UCI for discussions of the ideas expressed in the paper and for discussions on related systems such as FOIL and FOCL. Thanks also to Yousri El Fattah, who participated in the initial stages of this research.

Constraints for Predicate Invention[1]

Ruediger Wirth
Paul O'Rorke

Department of Information and Computer Science
University of California, Irvine, CA 92717
United States of America

Abstract

This chapter describes an inductive learning method that derives logic programs and invents predicates when needed. The basic idea is to form the least common anti-instance (LCA) of selected seed examples. If the LCA is too general it forms the starting point of a gneral-to-specific search which is guided by various constraints on argument dependencies and critical terms. A distinguishing feature of the method is its ability to introduce new predicates. Predicate invention involves three steps. First, the need for a new predicate is discovered and the arguments of the new predicate are determined using the same constraints that guide the search. In the second step, instances of the new predicate are abductively inferred. These instances form the input for the last step where the definition of the new predicate is induced by recursively applying the method again. We also outline how such a system could be more tightly integrated with an abductive learning system.

[1]This is a revised and expanded version of Wirth, R., & O'Rorke, P. (1991) Constraints on predicate invention. In L. A. Birnbaum, & G. C. Collins (Eds.), *The Eighth International Workshop on Machine Learning* (pp. 457–461). Evanston, IL: Morgan Kaufmann. Research supported in part by National Science Foundation Grant Number IRI-8813048, Douglas Aircraft Company, and the University of California Microelectronics Innovation and Computer Research Opportunities Program.

Contents

List of Figures

1 Introduction

In recent years there has been increasing interest in systems that induce first order logic programs. The approach of inverting resolution (Muggleton & Buntine, 1988; Rouveirol & Puget, 1989; Wirth, 1989) is particularly interesting because it offers a way to extend the vocabulary by inventing new predicates. However, the first implementations were too inefficient to be useful for larger applications.

Quinlan's FOIL (Quinlan, 1990) was an advance towards more efficient induction algorithms for first order languages. Subsequently, Muggleton & Feng (Muggleton & Feng, 1990) presented a new system, called GOLEM, which is based on inverse resolution and which is also able to process large numbers of examples. But, despite their efficiency these two systems are highly dependent on the vocabulary and the form of examples that are given in advance. They cannot extend their vocabulary.

This paper describes an attempt to overcome this limitation. We propose a new way to construct a first-order theory which allows for natural incorporation of background knowledge and the invention of new predicates. The method, implemented in a system called SIERES, is based on a general-to-specific search guided by constraints on the form of clauses.

Unlike FOIL, which searches in a very unconstrained space, SIERES iteratively increases the space by looking at increasingly complex clauses. If it cannot construct a clause that covers the training instances in the current restricted space using known predicates only, SIERES tries to invent a new predicate. There are some strict and heuristic conditions on the new predicate. If the predicate can be constructed, SIERES continues to learn a general definition for it, abductively deriving new instances.

Existing methods for inventing new predicates, for instance in the framework or inverse resolution like CIGOL (Muggleton & Buntine, 1988) or LFP2 (Wirth,1989) invent new predicates in order to reformulate a given set of clauses aiming at a more compact or a more comprehensible representation. Compaction is used for two purposes, compressing the theory and generalizing it. While there is a close connection between data compression and generalization, we claim that in the case of predicate invention it is beneficial to keep them apart. Compaction remains an important criterion for evaluating hypotheses but for predicate invention it should not be the dominating one.

2 Integrating Abduction and Induction

This paper describes a novel learning method that integrates abduction and induction. Abduction is used to complete explanations and infer specific missing facts. Induction is used to invent clauses and predicates in order to extend the general theory and improve its explanatory power.

2.1 Learning Specific Facts by Synthesis

Existing abductive learning methods learn while using general theories to construct explanations of specific observations (O'Rorke, Morris & Schulenburg, 1990). The learning method is a form of abductive inference. Queries that do not ground out in known facts are treated as more or less plausible hypotheses on the grounds that if they were true they would complete explanations of the observations.

A simple abductive learning method called "synthesis" was proposed by Pople (Pople, 1973). Technically, the method works as follows. Given an observation and a theory expressed as Horn clauses, backward chain in search of a proof justifying the literals of the observation. If two queries are generated that are unifiable, unify them and assume that the resulting literal is true. Since it enables one to explain two observations with the same hypothesis, Pople justified this operation in terms of Occam's Razor. Note that Pople's synthesis operation is non-deductive, so this method of abductive learning is a form of knowledge-level learning. In other words, if a literal is added to the theory by synthesis, it enlarges the deductive closure of the theory.

2.2 Learning General Facts by Anti-Synthesis

Least general generalization *LGG* (Plotkin, 1970), or least common anti-instance *LCA* (Lassez, Maher & Marriot, 1988),[2] can be used in an abductive framework to learn interesting new literals that are generalizations rather than specializations of literals that appear in existing rules. Assuming that Q_1 and Q_2 are two queries that arise in explanations of the same or different cases, $Q = LCA(Q_1, Q_2)$ is a hypothesis that would explain Q_1 and Q_2.

This is a dual to Pople's synthesis operator; call it anti-synthesis. In synthesis, the queries Q_1 and Q_2 have to be unifiable. This is not necessary in anti-synthesis. The queries Q_1 and Q_2 could be ground literals involving the same predicate symbol but different arguments. In synthesis, the queries unify to a new literal (their most general common instance). The queries both subsume this new literal. In anti-synthesis, the new literal Q is the least general common anti-instance of Q_1 and Q_2. It subsumes Q_1 and Q_2 but different substitutions might be used to get each instance. Like synthesis, anti-synthesis leads to new literals that improve the coherence of explanations.

2.3 Learning New Clauses

The LCA of abductive hypotheses serves as the initial candidate in our search for a clause that would enable us to complete an explanation. Assuming that the missing clause is

[2]Plotkin's least general generalization is equivalent to Lassez et al's least common anti-instance. Here we use the term LCA because it captures the meaning more precisely.

applicable to a set of abductive hypotheses, the head of the clause must be unifiable with each hypothesis, so it *must* be a generalization of these hypotheses. Unfortunately, the LCA of the abductive hypotheses is often overly general. During the generalization process important connections between input and output arguments are often lost. In this case, we specialize the learned clause by adding literals to its body such that the missing connections are restored. This enables us to acquire missing clauses other than unit clauses.

There are different ways to specialize a clause (e.g. Kietz & Wrobel, 1991; Quinlan, 1990; Shapiro, 1983) and different ways to constrain the search. In the next section we describe a method using a novel combination of constraints.

3 The Method

In the following description of the learning task and our method, we use the terminology of logic programming (Lloyd, 1987).

3.1 The Task: Learning New Clauses

Given:

- background knowledge P and
- a set of initial goals (training instances) $E = \{E_1, \cdots, E_n\}$ that follow from an unknown target program $P_{target} \supset P$, but not from P

the learning goal is to construct a set of definite clauses C_{target} such that

$$P \cup C_{target} \vdash_{SLD} E.$$

In other words, we want to extend a given theory to cover new examples.

3.2 Strict Constraints on New Clauses

Let us assume we are in a state with goals $\{E_1, \cdots, E_n\}$[3] where none of the clauses of the current program P is applicable to any of the E_i. We have to generate a new clause in order to complete the proof.

Assuming that there is exactly one clause missing, there are two strict constraints on this new clause:

[3]These goals could be either abductive hypotheses as described in the previous section or teacher provided training instances as in the usual inductive learning situation.

- Its head has to be unifiable with all the E_i.
- The clause has to produce the proper bindings for the output variables.

These two strict constraints form the basis of our method. Since the head of the clause must be unifiable with all the instances it must be a common anti-instance and must be at least as general as the least common anti-instance (LCA). Often, the LCA is too general. The only way to specialize it is to view it as a unit clause and specialize this unit clause by adding literals.

Usually, overgeneralizations are discovered using negative examples. But there is a syntactic way to identify some important cases of overgeneralization which is is related to the second constraint above. If the input/output behavior of the target predicate is given, for example in the form of *mode declarations* (Shapiro, 1983), unbound output variables indicate overgeneralization and, even more importantly, provide guidance to the specialization process.

Example: `reverse/2`. Let us assume we have a mode declaration `reverse(+,-)` specifying that the first argument is an input while the second argument is the output.[4] Now, let us look at the following three unit clauses, which could be the LCAs of sets of instances.

```
reverse([A],[A]).
reverse([A,B],[B,A]).
reverse([A|B],[C|D]).
```

The first two are correct according to the intended meaning of `reverse/2`. For any input list containing one or two elements these two unit clause generate the correct reversed lists. The third one is overly general. In it, there is no connection between the input and output arguments at all. This predicate would be true for any two lists. If we view this unit clause as a procedure with the mode declaration specified above, the output variables would remain unbound because they do not also appear as input variables. These unbound output variables indicate overgeneralization. The need to bind them in the body of a clause provides guidance to the specialization process. The following definition of *critical terms* tries to capture this.

Definition: *Critical terms* of the head of a clause are

- output variables that do not appear in the input arguments
- input variables that do not appear in the output arguments
- terms whose arguments are critical variables

[4] We use this notation throughout the paper.

Example: Given mode declaration `reverse(+,-)`, the critical terms of `reverse([A|B],[C|D])` are the members of the set {`[A|B]`, `A`, `B`, `[C|D]`, `C`, `D`}.

Critical terms provide a focus of attention while searching for a literal to add to the clause and are used to determine arguments of the newly invented predicate.

3.3 Heuristic Constraints on New Clauses

In addition to these relatively strict constraints, we employ heuristic constraints, in other words a representational bias, on the types of clauses that are to be learned. These additional constraints serve to prune the search space and provide information necessary for the invention of new predicates.

In meaningful clauses, the literals in the body are usually not independent of each other but share at least some variables. This dependency can be used to partially order the literals in a clause.

Definition: A literal L_2 *depends on* a literal L_1 if

- they share a variable V where
- V is an output variable in L_1 and
- V is an input variable of L_2.

Example: In a form of `reverse/2` defined:

```
reverse([A|B],[C|D]):-
        reverse(B,E),
        add_last(E,A,[C|D]).
```

with the mode declarations `reverse(+,-)`, `add_last(+,+,-)` the dependencies are as shown in figure 1. The tail of the input list in the head is passed to the first literal of the body as an input. The output of this recursive call is passed as an input to the final literal of the body. This literal also takes the first element of the initial input to the head and its output is passed back to the head as the output computed by the clause.

Example: In the form of merge sort defined:

```
merge_sort([A|B],[C|D]) :-
        split([A|B],E,F),
        merge_sort(E,G),
        merge_sort(F,H),
        merge(G,H,[C|D]).
```

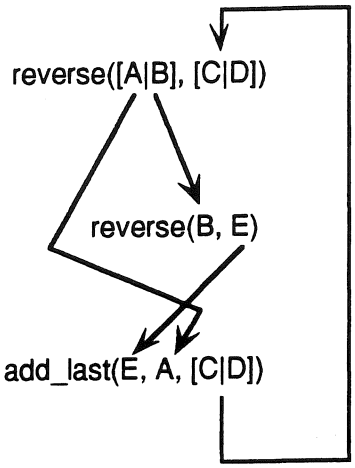

Figure 1: Dependencies in `reverse/2`

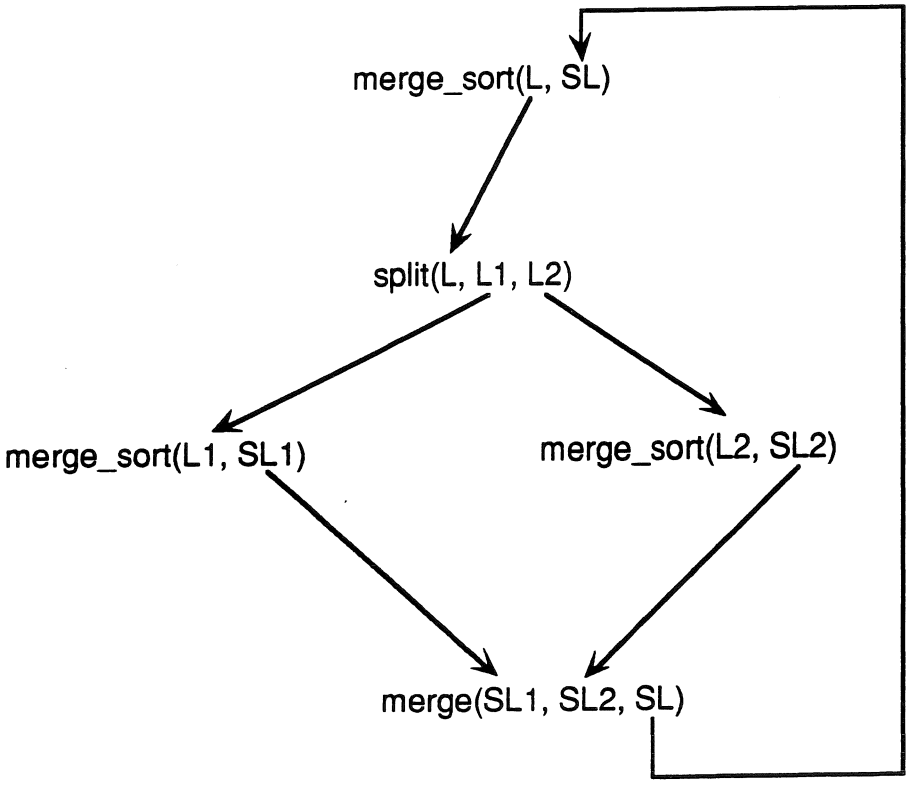

Figure 2: Dependencies in `merge_sort/2`

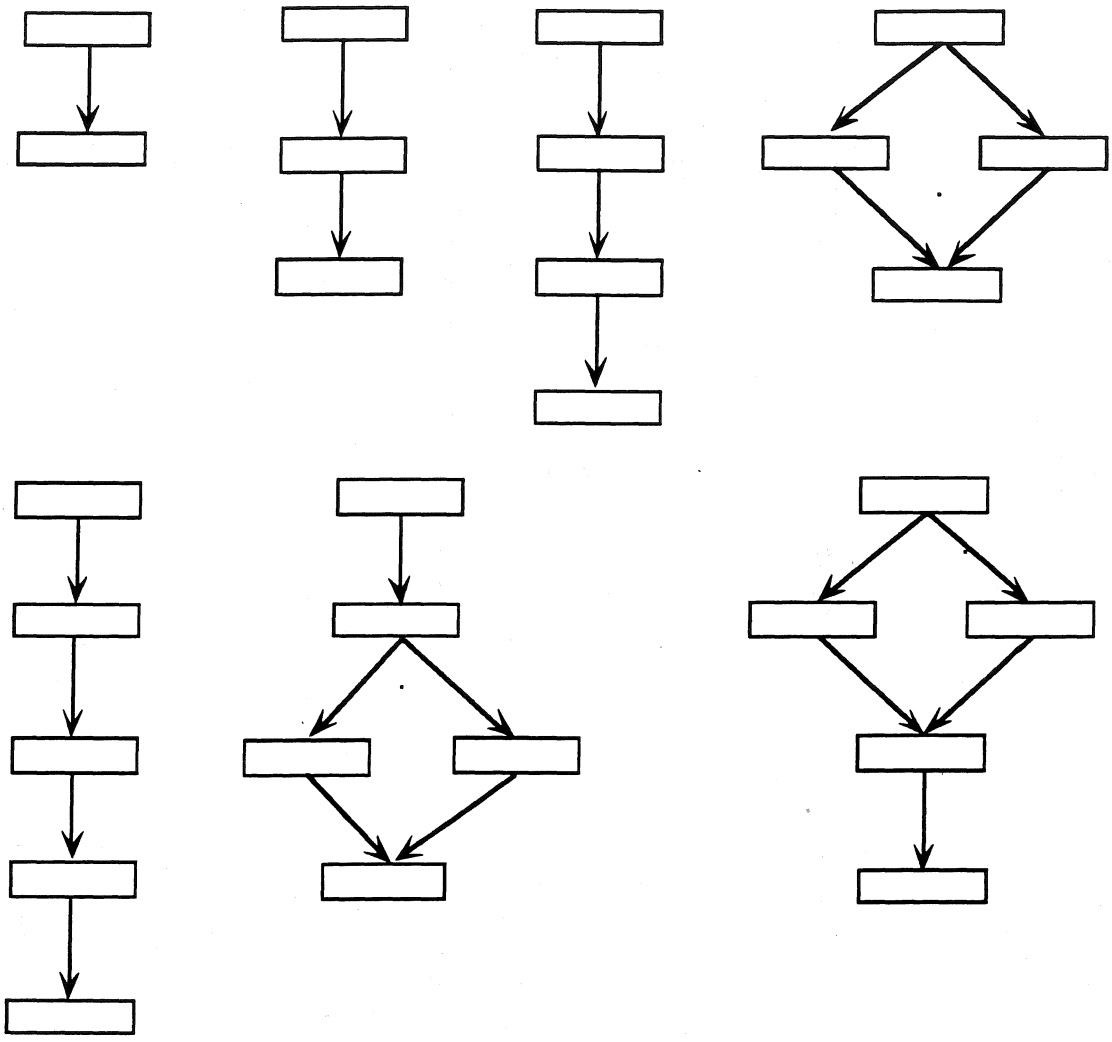

Figure 3: **Argument dependency graphs.** The boxes represent literals where the root is the head of the clause. The arrows indicate the dependencies between these literals.

with the mode declarations `merge_sort(+,-)`, `split(+,-)`, `merge(+,+,-)` the dependencies are as shown in figure 2.

SIERES is provided with a set of argument dependency graphs (figure 3), a kind of schemata. These argument dependency graphs are not detailed templates for the clauses, as for instance the rule models of (Kietz & Wrobel, 1991), but specify the dependency relationships that are allowed. Input terms of literals in the body must be subterms of input terms of the head or subterms of output terms of previous body literals. At least one input argument of each body literal must be a subterm of one output argument of the immediately preceding body literal. The output arguments of the last literal must bind all output variables in the head that are not yet bound. The restriction to subterms of previous terms serves to avoid an infinite set of possible terms.

These dependencies are used to guide the general-to-specific search. Muggleton & Feng (1990) use such dependencies for detecting relevant literals in a specific-to-general search.

3.4 Algorithm

As mentioned in the description of the task, the algorithm is given as input background knowledge and training examples. In addition,the algorithm is given mode declarations of all known predicates and a sequence of argument dependency graphs. The output of the algorithm is a set of clauses C such that the examples follow from the new clauses and the background knowledge.

The basic idea of the method (figure 4) is as follows. First, the LCA of the training instances is formed. If this is too general, a search for more specific clauses is conducted, subject to constraints that prevent the search from getting out of hand. New predicates are introduced as needed.

The current implementation of SIERES constrains search using mode declarations and a limited sequence of argument dependency graphs. Critical output variables provide a focus of attention while searching for a specialization of an overly general unit clause. The main goal of the search is to find a body that binds the critical output variables.

The first step in the algorithm is to select a subset of the examples such that this subset will be covered by just one clause.[5] From this subset, SIERES selects a few examples which serve as a seed for the remainder.

Then, SIERES selects an argument dependency graph. These graphs are ordered according to their complexity. SIERES starts with the simplest argument dependency graph and tries to find a clause that obeys the constraints of this graph. If such a clause cannot be found then a more complex dependency graph will be used for the next attempt.

[5]Currently, this part is limited to splitting simple recursive predicates into instances of the base case and of the recursive case.

SIERES keeps track of several copies of the argument dependency graph selected, one for the general clause which is supposed to be learned and one for each of the seed examples. The heads of all the clauses that are tried are the same regardless of the graph selected. The heads of the specific clauses are the seed examples, the head of the general clause is the LCA of the seed examples. The body of the specific clauses consist of facts from the background knowledge only. The general clause is the LCA of the specific clauses.[6] While constructing the body of the clauses there is a close interaction between the general and the specific clauses and information from both the general and the specific graphs is exploited.

At each step SIERES searches for each specific graph for a fact such that the constraints of the argument dependency graph are fulfilled and that the general clause, the LCA of the specific clauses, is legal, i.e. none of the literals has been generalized to a variable. Recall that input terms of literals in the body must be subterms of input terms of the head or subterms of output terms of previous body literals. Since this property must hold for both the general and the specific clauses in a coherent way, the general clause is used as a basis for determining potential input terms for the specific clause. The search space is considerably smaller this way. This interaction will become clearer in the example discussed in the following section.

If a general clause can be found that fulfills all the constraints, the algorithm terminates. If all but the last literal of the clause could be constructed, SIERES tries to invent a new predicate. The first problem is to determine the arguments of this predicate. As for the other body literals information from both the general and the specific clauses are used. The output terms of the new predicate are the critical output terms that are not yet instantiated in the general clause. The input arguments are determined using the critical input terms and the constraints from the argument dependency graphs. Critical input terms that have not yet been used in the general clause constructed so far are taken as input arguments. This can be justified by an Occam's razor argument. If these input terms were not used there would be no reason to provide them. More input arguments are derived from the literals directly preceding the last literal in the argument dependency graph. The longest output terms of the literal that share all their constants with the output terms of the new predicate are also taken as input arguments.

The next step is to find a definition for this new predicate. This is done in two steps. First, instances of the new predicate are generated. Second, these instances are given as input to a recursive call to SIERES. If a definition for this new predicate can be constructed, the algorithm will stop. Otherwise, it will backtrack and try alternative solutions.

[6]Of course, the LCA is not defined for clauses. But this is only a minor technical problem which can be overcome by viewing the specific clauses as terms. Then, the LCA is defined and the result can in turn be interpreted as a clause.

In order to construct instances of the new predicate, SIERES takes the general clause, applies it to all the known instances of the target predicate, and abductively infers instances of the new predicate to complete the proofs. The next section describes examples of learning sessions with SIERES.

4 Examples

In this section, we illustrate the method with examples. We give detailed descriptions of how SIERES learns `append/3`, `reverse/2`, and DeMorgan's law.

4.1 Learning append/3

Let us assume that we want to learn the definition of `append/3` with the mode declaration `append(+,+,-)`.[7] At the beginning we have a set of training examples including

```
append([s],[t],[s,t])
append([],[t],[t])
append([d,e,f],[g,h],[d,e,f,g,h])
append([e,f],[g,h],[e,f,g,h])
```

The seed examples may be `append([s],[t],[s,t])` and `append([d,e,f],[g,h],[d,e,f,g,h]` SIERES starts out by forming the LCA of the seed examples

```
append([A|B],[C|D],[A|E])
```

but this clause is not acceptable because it does not produce the correct answers when applied to the initial goals. The query `append([s],[t],X)` yields `append([s],[t],[s|E])` and the query `append([d,e,f],[g,h],X)` yields `append([d,e,f],[g,h],[d|E])`. These answers are overly general because they contain unbound output variables. Any instantiations of these variables would make the goals true, e.g. `append([s],[t],[s,applepie, honeypot])` would be provable. All the variables in the clause except `A` are critical. The critical output variable `E` is especially important.

SIERES searches for a specialization of the clause, starting with the next simplest argument dependency graph consisting of two literals. It initializes a general explanation and two specific explanations for the training instances. SIERES assumes that the output argument of the body literal has to be the critical output term `E` in the general graph and its corresponding instantiations in the specific graphs. Furthermore, the input arguments for the body literal have to be selected from the subterms of `[A|B]` and `[C|D]`

[7] If we also want to consider append in different modes, we could simply add the corresponding mode declarations. SIERES would treat the different versions of `append/3` as different predicates.

and their instantiations. Next, SIERES searches for predicates that could fit into these explanations under these constraints. In the background knowledge, it finds the facts `append([],[t],[t])` and `append([e,f],[g,h],[e,f,g,h])`, which would complete the specific clauses. By forming the LCA of these clauses SIERES finally obtains the clause

```
append([A|B],[C|D],[A|E]):-
        append(B,[C|D],E).
```

4.2 Learning reverse/2

This example employs abductive inference andrequires the invention of a new predicate. Let us assume that we are given a set of instances of **reverse/2** including

```
reverse([1,2,3],[3,2,1])
reverse([2,3],[3,2])
reverse([a],[a])
reverse([],[])
```

and that the seed examples are `reverse([1,2,3],[3,2,1])` and `reverse([a],[a])`. The LCA of the seed examples is `reverse([A|B],[C|D])` with the critical input terms {`[A|B],A,B`} and with the critical output terms {`[C|D],C,D`}.

SIERES starts with the simplest argument dependency graph consisting of two literals but cannot find a body literal that forms a legal solution. If it picked **reverse** it would produce a useless recursion where the recursive call were identical to the head. If it introduced a new predicate the arguments would be the same as for **reverse** and therefore no gain.

Next, SIERES tries to find a clause consisting of three literals (see figure 5). The head of the clause is again `reverse([A|B],[C|D])`. The input arguments of the first literal of the body of the general clause have to be selected from the set {`[A|B], A, B`}. SIERES looks up the substitutions for `[A|B]` for the seed examples and looks for facts that unify with `reverse([1,2,3],X)`, `reverse([a],Y)`. Such facts could form the first body literal of the specific clauses. The only facts it can find are the same as the heads and thus not suitable. Then it tries the substitutions for `A` but cannot find any because they are not of the proper type.

Finally SIERES tries the substitutions for `B` and looks for facts that unify with `reverse([2,3],X)` and `reverse([ ],Y)`. This time the search is successful and SIERES augments the specific clauses to

```
reverse([1,2,3],[3,2,1]) :-
     reverse([2,3], [3,2]),
     ???
```

and

```
reverse([a],[a]) :-
     reverse([], []),
     ???
```

The general clause is adapted by forming the LCA

```
reverse([A|B],[C|D]) :-
     reverse(B,E),
     ???
```

The argument dependency graph requires that one input argument of the last literal must be `E` and its instantiations. SIERES might find two facts for `reverse/2` with these input argument but their output arguments would not meet the additional requirement of binding all critical output terms. Thus, no known predicate fits and SIERES invokes he invention procedure.

The output arguments for the new predicate are the critical output terms that are not yet bound, i.e. `[C|D]`. The input arguments are the output terms of the preceding literals, i.e. `E` and all critical input terms that have not yet been used, i.e. `A`. In our example this leads to the clause

```
reverse([A|B],[C|D] :-
     reverse(B,E),
     new(A,E,[C|D]).
```

This definition is not complete because `new/3` is not yet defined. SIERES takes the rule above and applies it to all the known instances of `reverse/2`. Assuming that these instances are provable using this clause, SIERES abductively infers instances of `new/3` to complete the proofs. These instances will then be used to derive a definition of the new predicate.

Using the same mechanism as described above, SIERES derives the following definition for `new/3`:

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

4.3 Learning DeMorgan's Law

This example also requires the invention of a new predicate and illustrates the role of critical terms in a slightly different context. The goal is to learn the definition of a predicate `equiv/2` which implements DeMorgan's law for an arbitrary number of terms. The first argument of `equiv/2` is the input argument and is a negated conjunction. This expression is to be transformed into an equivalent disjunction of negations.

We start out with the following instances.

```
equiv(not(and([a])),or([not(a)]))
equiv(not(and([a,b])),or([not(a),not(b)]))
equiv(not(and([c,d,e])),or([not(c),not(d),not(e)])
```

The LCA is `equiv(not(and([A|B])),or([not(A)|C]))` but it is too general so SIERES seeks to specialize it using the next simplest argument dependency graph. SIERES initializes general and specific clauses as in the previous examples. The critical terms are `B` and `C`. Note that for the definition of `equiv/2` there is no need to consider `A` or the function symbols `and`, `or` and `not`. The definition of critical terms takes care of this into account.

SIERES is unable to complete these clauses using existing predicates so it invents a new predicate with the critical terms as the arguments. SIERES then completes the special explanations *by assuming the following*:

```
new13([],[])
new13([b],[not(b)])
new13([d,e],[not(d), not(e)])
```

The general clause corresponding to these specific clauses is

```
equiv(not(and([A|B])),or([not(A)|C])) :-
     new13(B,C).
```

This clause is acceptable provided that SIERES can construct a general definition for `new13/2` that also helps explain additional instances of `equiv/2`.

In order to learn a definition for `new13/2`, SIERES first needs to construct more instances. This can be done by applying the definition of `equiv/2` to different instances, e.g. `equiv(not(and([d,e])),or([not(d),not(e)]))`. This way, SIERES automatically constructs a training set for the new predicate, which can be used to learn its general definition.

Ultimately, SIERES learns the following program.

```
equiv(not(and([A|B])),or([not(A)|C])):-
     new13(B,C).

new13([],[]).
new13([A|B],[not(A)|C]):-
     new13(B,C).
```

5 Related work

SIERES is similar to FOIL (Quinlan, 1990) in that both construct clauses by searching for the best additions to the body of a partially constructed clause. FOIL, however, performs a search based solely on information theoretic heuristics. FOCL (Pazzani et al. 1991), an extension of FOIL, uses (possibly incorrect) background knowledge as a hint which predicates might be related to each other and to support the information gain heuristic's choice of the relevent predicates. Disjunctive definitions follow naturally from FOIL's and FOCL's control structure. FOIL and FOCL differ from SIERES in their restriction to function free clauses and their inability to extend the vocabulary automatically. Additionally, FOCL has serious problems with recursive definitions.

Our method is also related to RDT (Kietz & Wrobel, 1991), which also performs the general-to-specific search considering more and more complex clauses. RDT uses rule models to specify the shape of the clauses. Rule models are much more specific than argument dependency graphs in that they specify exactly the variables of the literals. There is no handle for predicate invention. Argument dependency graphs are more flexible because the dependencies between literals are expressed at a more abstract level which provides the constraints for the arguments of the new predicates.

GOLEM (Muggleton & Feng, 1990) constructs a least generalization with background knowledge and reduces this potentially huge clause according to several restrictions of the hypothesis space. Mode declarations and argument dependencies are used to generalize clauses. All literals that do not fit in a graph structure are deleted. GOLEM tries to avoid the overgeneralization problem with *LCA* by taking background knowledge into account and uses the *RLGG* (Muggleton & Feng, 1990). SIERES recovers from the overgeneralization of the *LCA* by using the background knowledge to specialize it.

Both FOIL and GOLEM as developed so far, have no real handle for inventing new predicates. FOIL has no notion about the processing of the terms of the clauses. All terms, i.e. variables in FOIL's case, are equal. Consequently, there is no criterion for choosing which variables to use in a new predicate. The information gain heuristics does not apply because nothing is known about the predicate to be invented.

GOLEM inverts a whole sequence of resolution steps. If it would start considering new predicates at each step without any hint what they are supposed to be good for, the

search space would explode.

The main advantage of SIERES compared to GOLEM or FOIL and FOCL is its ability to invent new predicates. In this respect, SIERES adds aspects from CIGOL (Muggleton & Buntine, 1988) and LFP2 (Wirth, 1989). However, the reason for introducing new predicates is different. In CIGOL and LFP2 the new predicates are introduced to compress the program. Both systems essentially check the knowledge base whether the introduction of a new predicate could result in a more compact representation. The invention procedure is not constrained by what the new predicate is supposed to be good for. Experience with LFP2 showed that this method leads to situations where a definition of an invented predicate was not general enough and had to be "re-invented" in a slightly different way. At the same time many interesting and useful predicates which could have been suggested by the operators did not pass the compaction filter.

CLINT-CIA (De Raedt & Bruynooghe, 1989) uses second order clause schemata to suggest useful new concepts to the user. These new concepts are not needed to prove anything new. Their purpose is to make theories more comprehensible and to allow the system to use a simpler learning bias. In this respect, CLINT-CIA is similar to LFP2 but different from SIERES.

The invention procedure in SIERES is goal-directed or demand-driven (Wrobel, 1988). This goal direction provides the context for the invention. The argument dependency relations and the critical terms both focus the search and specify what the new predicate is supposed to do.

Another distinguishing feature of SIERES is that it does not rely on a human oracle to evaluate the invented predicates as CIGOL, LFP2, and CLINT-CIA do. It only requires that the new predicate must allow the instances of the target predicate to be proved.

6 Current Status, Limitations, and Future Work

The ideas described in this paper have been implemented in an experimental system which has been tested on several logic programs including `reverse/2`, `append/3`, DeMorgan's law, and the following definition of merge sort.

```
merge_sort([A|B],[C|D]) :-
        split([A|B],E,F),
        merge_sort(E,G),
        merge_sort(F,H),
        merge(G,H,[C|D]).
```

As a consequence of the current relatively tight argument dependency constraints, `quick sort` and `reverse/2` as defined below are not learnable. In both clauses the

second argument of append is a term constructed from terms stemming from different literals.

```
quick_sort([H|T],Sorted) :-
        partition(H,T,L1,L2),
        quick_sort(L1,SL1),
        quick_sort(L2,SL2),
        append(SL1,[H|SL2],Sorted).

reverse([A|B],[C|D]) :-
        reverse(B,E),
        append(E,[A],[C|D]).
```

This suggests that the current constraints may be too restrictive. So one of our aims is to explore variations on the constraints looking to improve the space of learnable concepts while avoiding combinatorially explosive search.

More important research issues include the following. The current method learns one clause at a time and is not yet capable of learning arbitrary disjunctive definitions. Stahl (1991) developed a method for splitting the training set such that each subset is very likely to be covered by just one clause. The next implementation of SIERES will incorporate this technique. It will also abandon the fixed sequence of argument dependency graphs and employ a more flexible, heuristic general-to-specific search. Together, these two improvements are expected to overcome this current limitation.

The use of LCA as a basis for learning raises important questions about the quality of the seed examples. LCA is very susceptiple to chance regularities such as coincidentally having the same constant at the same position of the terms that are to be generalized. It is not yet fully understood what makes good examples for LCA but some heuristic criteria are available. One heuristic is to avoid having the same constants in different examples whenever possible. If the examples contain recursive data structures as terms good seed examples should have terms of different depths and should have both shallow and deep terms. Ling (1991) discusses the problem of good examples and representative data sets from a different perspective.

So far the system is mostly inductive. Abduction is restricted to assuming new facts about the new predicate to be invented. This is fairly safe because the system itself defines the meaning of the predicate and it can assume whatever it wants as long as the target predicate is correctly defined. However, the abductive aspects could be strengthened at various points. Currently, SIERES makes a closed world assumption while constructing the specific clauses.[8] This assumption is not always justified, especially if SIERES is

[8]This closed world assumption does not extend to the meaning of the target predicate. As in any

integrated with an abductive system as outlined in section 2. Relaxing the closed world assumption would allow SIERES to assume new facts for the target predicate.

We believe that SIERES provides a good framework for exploring interactions between inductive and abductive learning. So far we touched only a small part of this vast area. Further investigations have to reveal its true potential.

7 Conclusion

We have described a learning method, implemented in a system called SIERES. The method integrates abduction and induction in a natural way and provides a good starting point for further investigations Constraints provided by syntactic least common anti-instance, critical terms, and argument dependency graphs focus a general-to-specific search for new clauses. These constraints are also exploited for predicate invention.

The method invents new predicates in three steps. In the first step, the need for a new predicate is discovered and its arguments are determined. The second step uses abduction to infer more instances of the new predicate. The third step uses these instances for inducing a general definition of the new predicate.

Acknowledgments

We thank Mike Pazzani, Dennis Kibler and the graduate students of the AI & ML community at UCI for discussions of the ideas expressed in the paper and for discussions on related systems such as FOIL and FOCL. Thanks also to Yousri El Fattah, who participated in the initial stages of this research.

inductive learner, the final definition of the target predicate typically covers facts that were not given as training instances.

References

DeRaedt, L., Bruynooghe, M. (1989). Constructive Induction by Analogy: A method to learn how to learn? In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 189-200). Montpellier: Pitman.

Kietz, J.-U., & Wrobel, S. (1991). Controlling the complexity of learning in logic through syntactic and task-oriented models. This volume.

Lassez, J.-L., Maher, M. J., & Marriot, K. (1988). Unification revisited. In J. Minker (Eds.), *Foundations of deductive databases and logic programs* (pp. 587-626). Los Altos, CA: Morgan Kaufmann.

Ling, C.X. (1991). Logic Program Synthesis from Good Examples. This volume.

Lloyd, J. W. (1987). *Foundations of logic programming* (2nd ed.). Berlin: Springer-Verlag.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 256-269). Ann Arbor, MI: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceeedings of the First Conference on Algorithmic Learning Theory* (pp. 1-14). Tokyo, Japan: Ohmsha.

O'Rorke, P., Morris, S., & Schulenburg, D. (1990). Theory formation by abduction: A case study based on the chemical revolution. In J. Shrager, & P. Langley (Eds.), *Computational Models of Scientific Discovery and Theory Formation* (pp. 197-224). San Mateo, CA: Morgan Kaufmann.

Pazzani, M., Brunk, C., Silverstein, G. (1991). An information-based approach to combining inductive and explanation-based learning. This volume.

Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine Intelligence* (pp. 153-163). Edinburgh: Edinburgh University Press.

Pople, H. E. (1973). On the mechanization of abductive logic. *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 147-152). Stanford, CA: Morgan Kaufmann.

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

Rouveirol, C., & Puget, J. F. (1989). A Simple Solution for Inverting Resolution. In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 210-210). Montpellier: Pitman.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging.* Cambridge, MA: The MIT Press.

Stahl, I. (1991). *Induktion von disjunktiven Konzepten.* Diplomarbeit, University of Stuttgart (in German)

Wirth, R. (1989). Completing logic programs by inverting resolution. In K. Morik (Ed.), *Proceedings of the Fourth European Working Session on Learning* (pp. 239-250). Montpellier: Pitman.

Wrobel, S.(1988). Automatic representation adjustment in an observational discovery system. In D. Sleeman (ed.). *Proc. of the 3rd European Working Session on Learning* (pp. 253-262). Glasgow: Pitman.

sieres($Predicate$,$Examples$, $Theory$, $ADGs$):
  Until $Examples$ provable from $Theory$
    Select $Examples' \subseteq Examples$
    Select $Seed \subseteq Examples'$
    Select an argument dependency graph $ADG \in ADGs$
    Let G be an instance of ADG
    Set $head(G) = lca(Seed)$
    For all $E \in Seed$
        Let S_E be an instance of ADG
        Set $head(S_E) = E$
    Subject to constraints associated with ADG
        For all $E \in Seed$
            instantiate $body(S_E)$ with facts from $Theory$
        $G = lca(S_E)$
    If the last literal in $body(G)$ remains uninstantiated
        $NewPred = new\_predicate(G)$
        $Examples'' = generate\_examples(NewPred, Examples, Theory, G)$
        sieres($NewPred, Examples', Theory, ADGs$)

Figure 4: Pseudo-code for SIERES

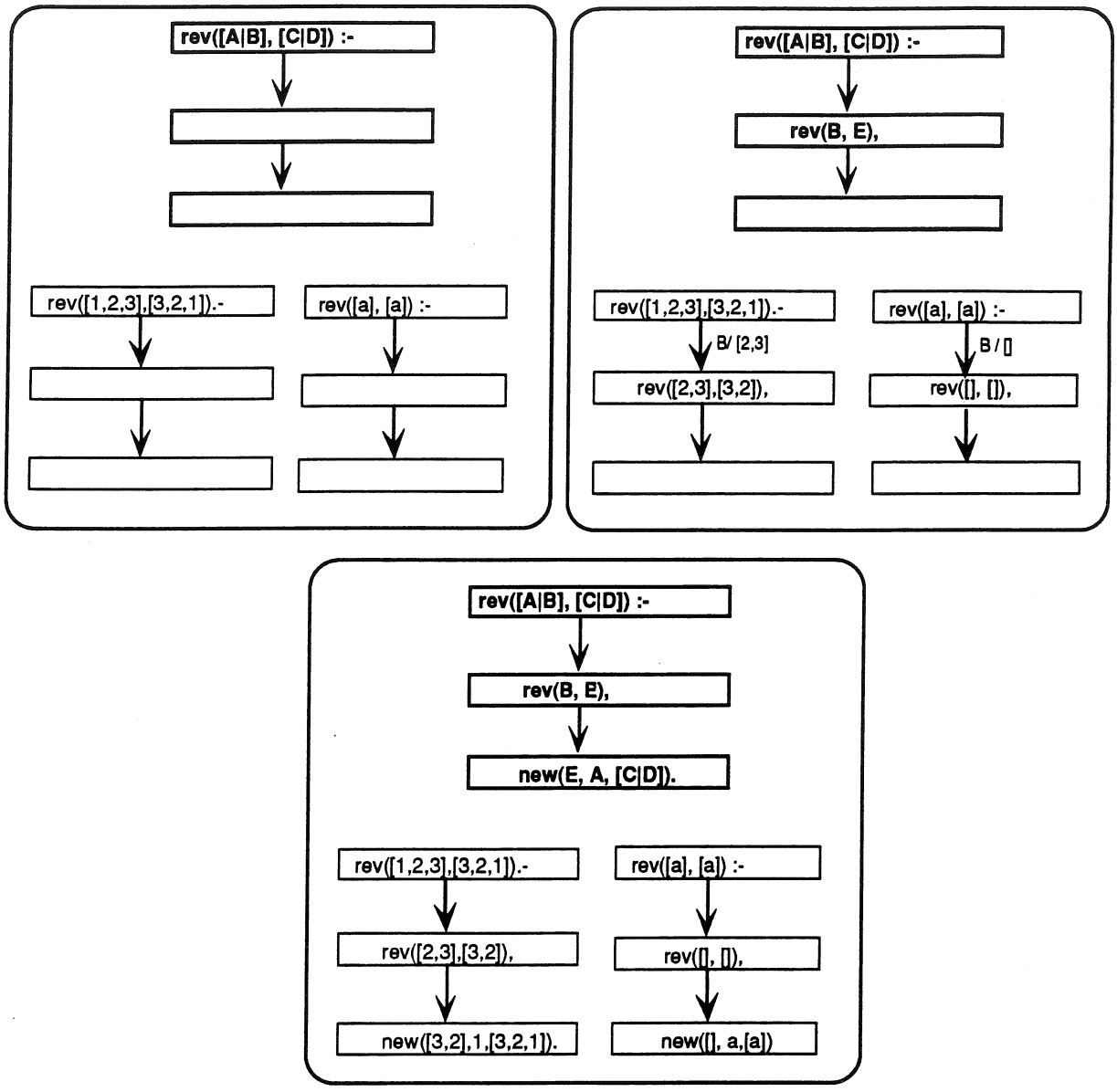

Figure 5: Learning `reverse/2`