Machine Learning 3: 45–77, 1988
© 1988 Kluwer Academic Publishers, Boston – Manufactured in The Netherlands

Learning by Failing to Explain: Using Partial Explanations to Learn in Incomplete or Intractable Domains

ROBERT J. HALL (RJH@WHEATIES.AI.MIT.EDU)
Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.

(Received: March 15, 1987)
(Revised: February 8, 1988)

Keywords: Learning by failing to explain, explanation-based learning, graph grammar, design

Abstract. Explanation-based learning depends on having an explanation on which to base generalization. Thus, a system with an incomplete or intractable domain theory cannot use this method to learn from every precedent. However, in such cases the system need not resort to purely empirical generalization methods, because it may already know almost everything required to explain the precedent. *Learning by failing to explain* is a method that uses current knowledge to prune the well-understood portions of complex precedents (and rules) so that what remains may be conjectured as a new rule. This paper describes *precedent analysis*, partial explanation of a precedent (or rule) to isolate the new technique(s) it embodies, and *rule reanalysis*, which involves analyzing old rules in terms of new rules to obtain a more general set. The algorithms PA, PA-RR, and PA-RR-GW implement these ideas in the domains of digital circuit design and simplified gear design.

1. Introduction

Every approach to machine learning depends, at least implicitly, on some set of constraining assumptions about the nature of the learning task. Empirical learning approaches, such as conceptual clustering algorithms (Michalski & Stepp, 1983), require that observed regularities in the data reflect the underlying causal laws of the domain. This assumption tends to be true in domains where training examples are relatively abundant. Another frequently-used constraining assumption is that the target concepts can be expressed in a limited (biased) language (Winston, 1975; Mitchell, Utgoff, & Banerji, 1983).

Explanation-based approaches to learning (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986) assume that examples or precedents can be explained in terms of the learner's theory of a domain. They then generalize the resulting explanation in order to find weaker preconditions under which the explanation still holds. These techniques have been applied to several widely different domains. For instance, Mooney and DeJong (1985) describe a system for learning schemata for natural language processing, whereas Mahadevan (1985) and Ellman (1985) apply similar techniques to the circuit domain. Mooney and Bennett (1986) give a domain-independent technique for generalizing hierarchically structured explanations, and Minton, Carbonell, et al. (1987) outline a general planning architecture that uses related techniques.

One limitation of explanation-based approaches is that they fail if the system cannot explain the precedent. As Mitchell et al. (1986) have pointed out, a system can fail to find an explanation for a variety of reasons, including incompleteness and intractability of the domain theory. To illustrate the former problem, consider an adder circuit that differs from a known design only in the implementation of one exclusive-or gate. It would be overly conservative to treat the entire structure as a new adder – it is really an old adder with a new exclusive-or gate. Since an explanation-based technique cannot completely explain the new adder, it cannot be applied to this generalization problem. A more versatile learner would explain those parts it could understand, such as the bit-slice architecture and the sum-with-carry structure of each bit. It would then see that it failed to explain the new adder only because it could not understand a particular subcircuit. Thus, the system could conjecture that its domain theory was missing an implementation rule for exclusive-or.

Another limitation of explanation-based approaches is that they can only produce rules that are provably correct (in their domain theories) independent of context. That is, they cannot learn logically incorrect rules that nevertheless have heuristic value. Explanation-based methods compile rules from the domain theory into more efficient ones, attempting to find the weakest preconditions under which the compiled rules should be applied. However, finding the weakest preconditions is a notoriously difficult problem, and instead the system may produce a logically correct but overly specific rule. In contrast, heuristic rules constitute plausible conjectures that are useful in some contexts but invalid in others. Most explanation-based learning methods cannot acquire such heuristic knowledge.

This paper describes an approach to learning in the presence of imperfect domain theories, which we call *learning by failing to explain*. The intuition behind this approach is that novices do have some useful explanatory knowledge, which they can use to isolate those aspects of a precedent they do not understand. For example, a student who can say, "I don't understand step five," is in better shape than one who says, "I don't understand this proof."[1] This approach does not distinguish between intractable and incomplete theories; each can lead to a failure to explain and thus is treated the same way. Furthermore, like all empirical learning methods, the current method can only form *plausible conjectures*. It does not address the issue of verifying the soundness and usefulness of these conjectures.

The constraining assumption underlying this approach is that precedents can be derived from a grammar of generally useful problem-solving operators. Such an assumption seems plausible in design domains, where designers build up a hierarchy of implementation techniques. It would be less plausible in a

[1]This work is motivated by intuitions about human learning, but is not intended to model such learning.

domain where the interesting properties come about through significantly random processes. The following section describes learning by failing to explain and its application to the design domain. Section 3 evaluates the approach, examining its generality and limitations, reporting some experimental results, and outlining directions for further research. Finally, Section 4 discusses relations to other learning work. Hall (1985, 1986a) goes into greater depth on the issues discussed in this paper.

2. Learning by failing to explain

This section presents one approach to learning by failing to explain. It begins with a description of the design domain and then examines issues of representation and performance within this domain. Finally, it considers two related learning methods – precedent analysis and rule reanalysis.

2.1 The design domain

I have tested these learning methods in the design domain, focusing principally on circuit design. The devices in this area fall naturally into a hierarchy, with more complex devices, such as adders and register banks, made from interconnections of less complex devices, like bit slices, flip-flops, and gates. I have also examined the design of simplified gear mechanisms, in which elements such as gears, sprockets, chains, and shafts are connected to implement transfer of rotational speed.

A model of the design skill must account in some way for at least the following competences.

- *Top-down design*: the ability to take a high-level specification of a device's function and refine it into a lower-level implementation.
- *Optimization*: the ability to take one device and replace a piece of it with some other piece so that the resulting device is functionally the same but improved in some way.
- *Design derivation (analysis)*: the ability to justify that some device performs a given function.
- *Analogical design*: the ability to solve a new problem in a way similar to some previously solved problem, or by combining elements of the solutions to several old problems.

This paper discusses learning in the context of a particular model of design knowledge, the design grammar. In order to justify use of this model of knowledge, Section 2.3 discusses how a design grammar supports these four competences, though design performance is not the primary focus of this research.

The learning task I will address involves the improvement of design competence with experience. This can be stated more formally:

- *Given:* A set of design rules that provide initial competence;
- *Given:* A set of *precedents* giving specifications and their implementations;
- *Find:* New design rules that, in combination with the initial rules, let one transform the specifications into their implementations.

Note that the above statement refers to *precedents* rather than *instances.*[2] I assume that each input is actually a complex combination of instances of many different rules. Thus, a large part of the learning problem lies in determining the boundaries between rule instances. This contrasts with most work on concept learning, in which each input corresponds to a single rule instance.

2.2 Representing designs and design knowledge

The principal epistemological assumption I make is that design knowledge can be divided into two components: knowledge about the analysis of designs and knowledge about their synthesis, with the latter being represented as a design grammar. This distinction is critical to understanding what is learned, and it is shared by many authors, including Mahadevan (1985) and Ressler (1984).

Analysis knowledge is that body of axioms, lemmas, theorems, proof techniques, etc., that lets one show that a design is correct or calculate its costs. This is a deductively sound knowledge base, which I take as given for present purposes. For example, analysis knowledge in the circuit world includes DeMorgan's laws and the laws of temporal logic.

Synthesis knowledge, on the other hand, is what enables the designer to put together (synthesize) implementations. It is a collection of tools and tricks that the designer uses in producing implementations. For example, one synthesis rule for circuits is "To implement NOR(a, b), use AND(NOT(a),NOT(b))."

In contrast to analysis knowledge, synthesis knowledge may contain heuristic rules that may not preserve functional equivalence in all contexts. For instance, one can multiply a number by two using a single bit shift, but one can not multiply a number by three this way. The single shift is an implementation of multiplication that works only when one of the inputs is two.

Synthesis knowledge is represented as rules having the form "structure X can sometimes implement functional block Y," where a functional block represents some constraint between its well-defined inputs and outputs. By structure, I mean an interconnection of functional blocks, where the interconnection represents the flow of data.

Each rule has the form LHS $\Longleftrightarrow$ RHS, where LHS denotes a single functional block and RHS describes a possible implementation for LHS. Structures are represented by directed graphs. Figure 1 graphically illustrates a typical rule. This may be represented algebraically as

$$[y = \mathrm{BUFFER}(a)] \Longleftrightarrow [y = \mathrm{NOT}(\mathrm{NOT}(a))].$$

Note that input and output correspondences are indicated by common variables in the two sides of the rule. Thus in the figure, the output variable y in the LHS is the same as the output variable y in the RHS.

In this representational scheme, a precedent consists of a pair of graphs, each representing an implementation of the same overall function. The intuitive

[2]This terminology is in accord with Winston, Binford, Katz, and Lowry (1983), who also use *precedent* to mean complexes of rule instances.

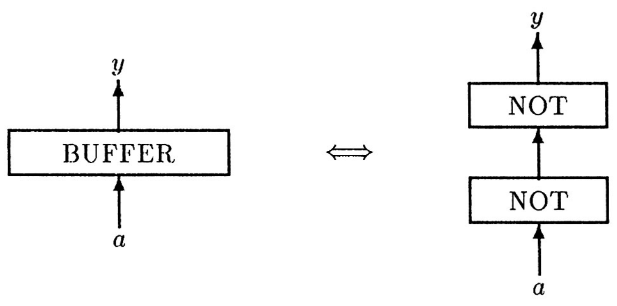

Figure 1. A design grammar rule. The LHS is to the left of the double-headed arrow, the RHS to the right. Input/output variable correspondences are represented by naming. Thus, the two *a* variables correspond and the *y* variables correspond.

meaning is that one is a high-level description of some device and the other is a particular implementation of the same device. Therefore, one of the graphs will be termed the *high-level* graph, the other the *low-level* graph.

2.3 Derivation in design grammars

Having formalized synthesis knowledge in terms of grammar rules, we will consider how such grammars can be used in the process of derivation. A *derivation* is a sequence of rule applications, starting from some graph (a structure description) and leading to a functionally equivalent graph. To apply a rule, the system must first verify that either the LHS or RHS of the rule matches a subgraph of the current graph. If this holds, then the system removes the matching subgraph from the current graph and installs a copy of the other side of the rule in its place. Analytic knowledge is then used to verify that the transformation maintains overall functional equivalence between the initial graph and the resulting graph. (When this is verified, the rule application is termed *allowable.*) Note that the system may apply rules either left to right or right to left, as long as the analysis component verifies that the transformation maintains overall correctness. If the system cannot verify correctness, the rule application is not allowed (and cannot be a part of any derivation).[3]

The four design competences can be seen as different applications of the basic mechanism of grammar derivation, as follows. Note that I have not implemented a complete performance design system to demonstrate these competences. Other authors have done work in this area, however, as noted below.

Top-down design. This involves taking a relatively high-level specification of the function of a device and successively refining it by choosing implementations of subfunctions. This process is modeled by derivations using rules exclusively left to right, that is, by always replacing the single LHS functional

[3]This research has not addressed the problem of verification using analytic knowledge, so when such verifications are needed, the system asks a human oracle.

block with the RHS implementation. Ressler (1984) takes this approach in generating operational amplifier designs.

Optimization. This involves taking one device and replacing one piece with another so the resulting device is functionally the same. An optimization step can be modeled by a right-to-left rule application, followed by a left-to-right rule application. For example, consider representing the idea that a synchronous adder circuit can sometimes be made faster by pipelining, that is, increasing the latency but decreasing the clock cycle time. Suppose a system has two implementation rules for the functional block SYNCHRONOUS-ADDER, one with high latency and short clock period, the other with low latency and long clock period. Using these rules, the system can recognize when one implementation is present as a subgraph of the current design stage and replace it with the functional block SYNCHRONOUS-ADDER. Then the block can be expanded using the other implementation.[4] Darlington (1981) uses a similar notion of transformation in order to semi-automatically optimize programs.

Design derivation (analysis). This involves justifying that some device performs some given function. This process can be viewed as the parsing problem for design grammars. Wills (1987) takes this view in her work on recognizing program structural cliches. Winston et al. (1983) take a similar view of analysis, but the grammatical structure is implicit in the knowledge base of precedents, and rules are extracted "on the fly" from analogous precedents.

Analogical design. This involves solving a new problem in a way similar to some problem that has already been solved, or by combining elements of the solutions to many old problems. This competence can be modeled by "re-running" a known derivation on a new design problem by: (1) finding a partial match between the problem specification and the initial specification of the known derivation, (2) applying those steps whose subgraphs are covered by the partial match (and which are still allowable), and (3) leaving out the other steps. This technique for controlling search has been explored by Steinberg and Mitchell (1984), and can be seen in STRIPS' use of macrops (Fikes, Hart, & Nilsson, 1972).

The learning task can now be stated in more technical terms. The system should conjecture general grammar rules to add to its design grammar so that the precedents have derivations in the augmented design grammar. That is, for each precedent, there should be a grammar derivation starting from the high-level graph and resulting in the low-level graph.

2.4 Precedent analysis

The distinction between analytic and synthetic design knowledge leads to two forms of explanation. Some previous work on explanation-based learning (e.g., Mahadevan, 1985) has focused on deductively valid explanations based on analytic knowledge. Others (e.g., Mooney & DeJong, 1985) have focused on heuristic synthetic knowledge, in which a design is explained by the sequence

[4]The issue of verifying that such a move improves efficiency is also beyond the scope of this work.

of rule applications used to derive it. I have taken the latter approach in my own work.

Now let us consider precedent analysis, a technique for constructing and learning from a partial explanation of a precedent. PA is a prototype computer program that implements this technique. The basic idea behind the algorithm is to first construct a maximal partial parse (derivation) of the precedent in terms of the existing design grammar. Next one prunes the parts of the precedent's graphs that are completely derived by the partial parse, leaving a new correspondence between a subgraph of the current high-level graph and a subgraph of the low-level graph. Finally, one forms two new rules (with identical LHSs) using this new correspondence.

Although there have been other approaches to parsing, both for languages generated by graph grammars (Brotsky, 1984) and by string grammars, these approaches cannot be applied to the problem of constructing partial parses for use by precedent analysis. This is because (1) the algorithm must produce reasonable partial parses when the input cannot be completely parsed; and (2) most existing parsers depend upon the fact that they parse context-free or context-sensitive languages, whereas design grammars can generate arbitrarily complex languages. In fact, the parsing problem for design grammars is uncomputable (Hall, 1985).

2.4.1 A precedent analysis scenario

Consider an example of this process. Suppose precedent analysis is given the following precedent:

$$\text{(a) } [y = Z^{-1}(\beta), \beta = \text{MUX}(K, \text{ZERO}, Z^{-1}(\text{XOR}(\alpha, c))), \alpha = \text{XOR}(a, b)]$$

$$\equiv$$

$$\text{(b) } [y = Z^{-1}(\beta'), \beta' = \text{AND}(K, \text{XOR}(Z^{-1}(\alpha'), Z^{-1}(c))), \alpha' = \text{XOR}(a, b)].$$

Figure 2 presents this information graphically, with (a) considered to be the high-level graph. Suppose further that the learner knows the design grammar rules shown graphically in Figure 3.

Using the knowledge in this miniature design grammar, the learner can already show that the MUX with ZERO on an input in Figure 2 (a) is equivalent to the AND box in Figure 2 (b). Apply the rules in the order given: expand the MUX, replace the AND-ZERO combination with ZERO, replace the resulting OR-ZERO combination with BUFFER, and finally implement the BUFFER as a single connection point. This partial derivation leads to the equivalence shown in Figure 4.

The precedent analysis system then reasons that the functionally corresponding parts of the two graphs have been explained by the partial derivation. This correspondence is indicated by the dashed lines in Figure 4. In this case, the fact that the original MUX-ZERO combination can be implemented by the AND is explained by the partial derivation. The learner knows the remaining (unmatched) subgraphs fulfill the same role in the overall function. It is then reasonable to conjecture that this role equivalence derives from some combination of unknown rules, and thus should be captured in the grammar.

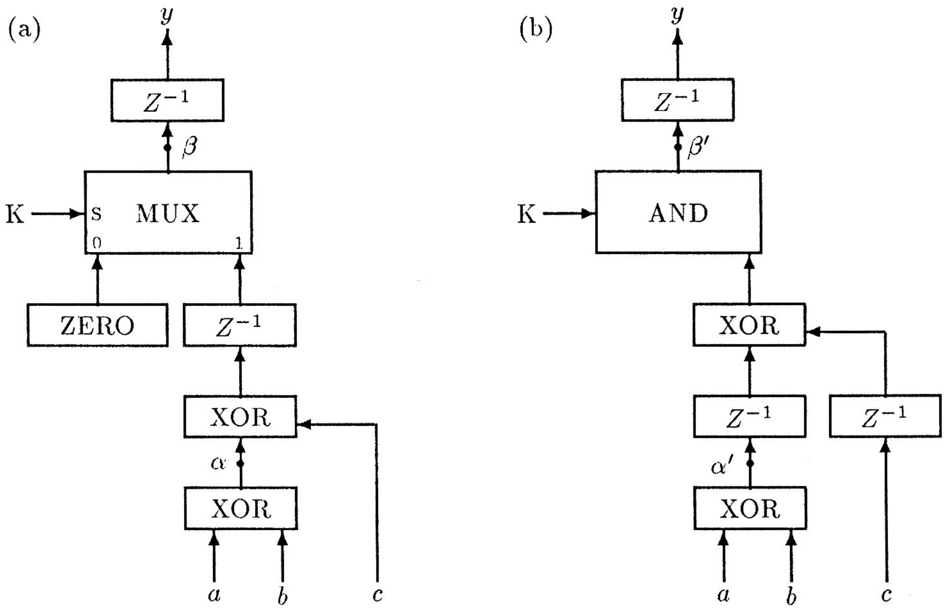

Figure 2. A precedent, consisting of a high-level graph (a) and a low-level graph (b). This example is somewhat contrived in order to more fully illustrate the parsing algorithm's heuristics. The Z^{-1} boxes represent a time delay of one clock cycle.

Since both subgraphs are more than single nodes, neither can be the LHS of a rule. Thus, the learner creates a new functional block type to stand for the common role[5] filled by the two subgraphs. This block then becomes the LHS of two rules shown in Figure 5, one whose RHS is the unmatched subgraph in Figure 4 (a), the other whose RHS is the unmatched subgraph in Figure 4 (b).[6] The variable correspondences are determined by the partial match.

Creating the appropriate name for the common role (function) filled by the two subgraphs can be a problem in that (1) the teacher will not share the same name for that role, and (2) the learner may already know another name for (what turns out to be) the same role. The present system ignores these problems by simply making up a new name each time. The problem of deducing that two roles are the same (from their implementations) requires sophisticated analytic reasoning beyond the scope of this research.

[5]In my terminology, "role" is a generalization of "function"; see Hall (1986b) for more discussion of roles.

[6]At the current stage of this research, creating two rules rather than one is no different in power from creating a single rule associating the two unmatched subgraphs. However, I anticipate that this will facilitate future research. The new LHS functional block can be given a natural interpretation as the *most restrictive common role* of the two RHSs.

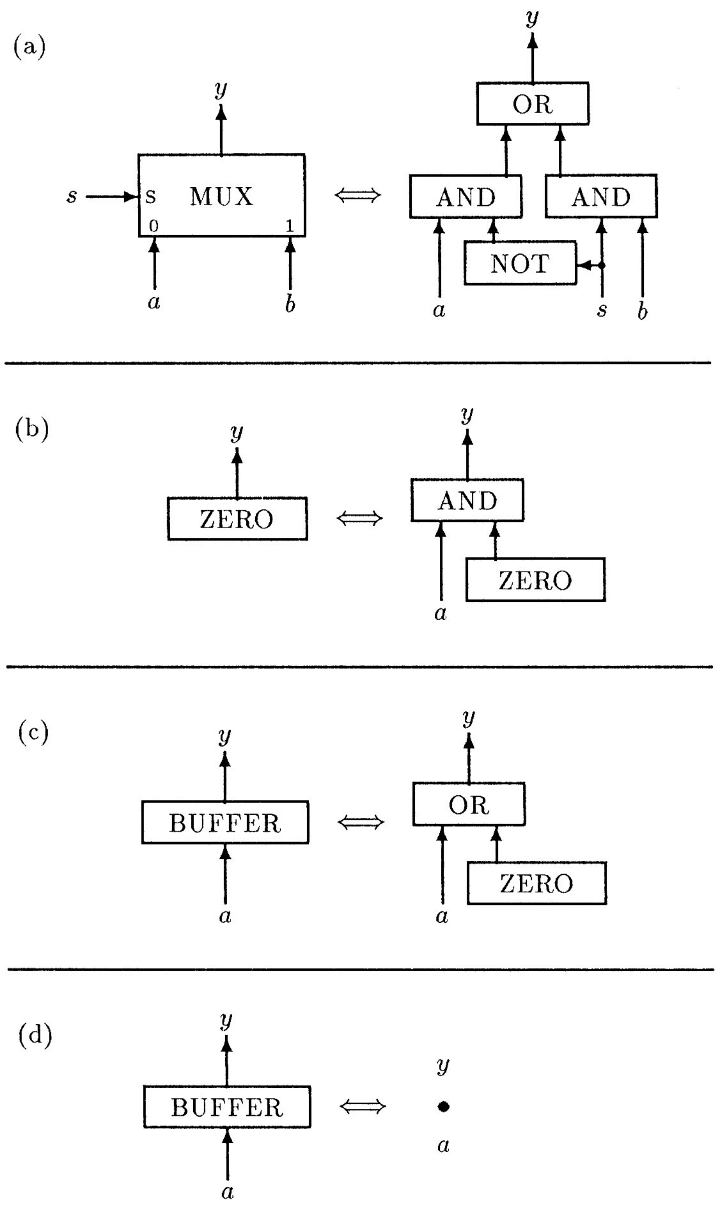

Figure 3. Four design grammar rules. Rule (a) is a standard multiplexor implementation, whereas (b) reflects the fact that AND of anything with zero is identically zero. Grammar rule (c) shows that OR of anything with zero is the identity (BUFFER) function, and (d) shows that the identity function may be implemented as a connection point.

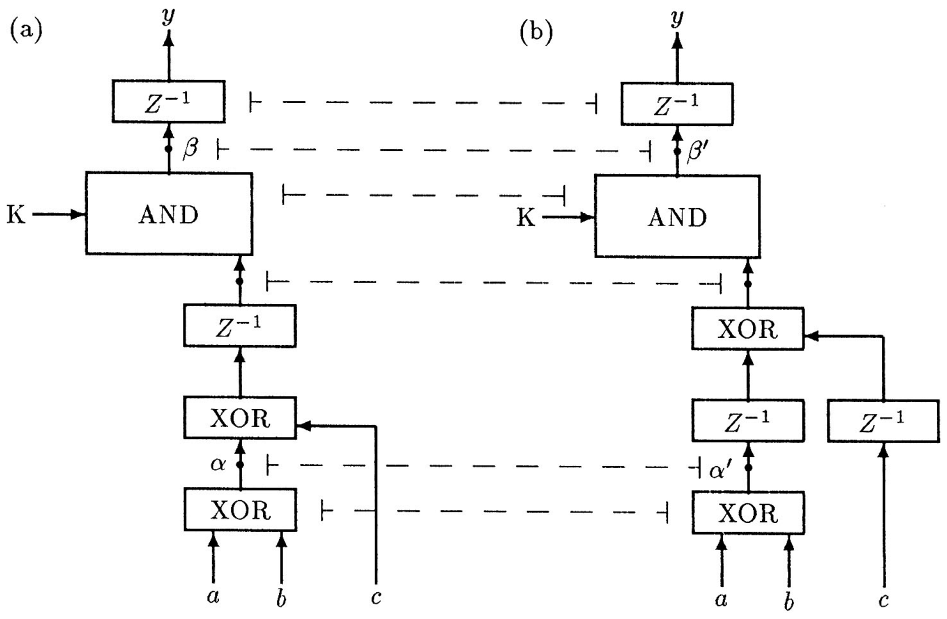

Figure 4. The precedent in Figure 2, after prior knowledge (Figure 3) has led to partial understanding, showing that the MUX-ZERO combination of the original is implemented by the AND box. The dashed lines indicate functional correspondences. Note that functional blocks correspond to functional blocks and connection points correspond to connection points. The Greek letters are for reference in the text only.

2.4.2 The precedent analysis algorithm

Now that we have seen an example of precedent analysis, let us consider the method in more formal terms. Table 1 presents a pseudo-code description of the PA algorithm, which implements a greedy approach to finding partial parses. The method searches for a partial derivation that, when applied to the high-level graph, results in a graph with as large a partial match as possible with the low-level graph. The partial match must initially match corresponding input and output variables as dictated by the precedent. Ideally, at any stage in its development, the partial match may only associate functionally equivalent nodes in the two graphs. The heuristic criteria for extending the partial match are given below. In the pseudo-code description, PM holds the current partial match, HLG holds the current high-level graph, and LLG holds the (unchanging) low-level graph of the precedent. The PA algorithm loops, calling the SEARCHFORPROGRESS algorithm shown in Table 2. Each time progress is found, the loop is executed again, starting from the HLG resulting from the previous iteration. When no more progress occurs, the resulting HLG is used to formulate a conjecture.

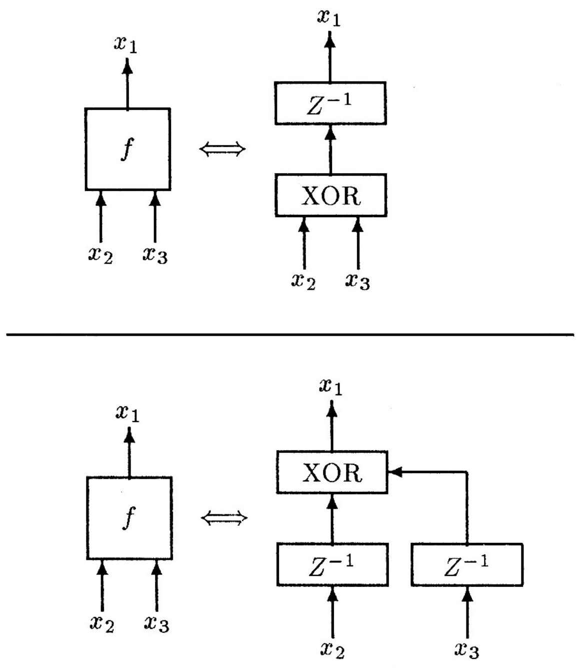

Figure 5. The grammar rules learned from the precedent in Figure 2. The system has extracted the role-equivalent subgraphs from the transformed precedent. Since neither subgraph was a single node, two new rules were formed. The variable correspondences (indicated by naming) were inferred from the partial match.

SEARCHFORPROGRESS looks ahead through all possible combinations of allowable rule applications to a fixed search depth, trying to find some combination that results in a meaningful extension to the partial match. The search is breadth-first through the space of rule applications, stopping at a depth dictated by the system parameter SEARCHDEPTH. When no further progress seems possible, SEARCHFORPROGRESS returns the smallest graph encountered in the last lookahead phase. The complexity of the subgraph isomorphism problem makes it desirable to have the graphs of grammar rules be as small as possible.

In the previous example, PA proceeds as follows. First, progress is found immediately (without any rule applications), because the match can be extended to include points α-α' and β-β'. Assuming SEARCHDEPTH = 4, SEARCHFORPROGRESS then searches breadth-first to depth four, resulting in Figure 4. Further progress is made by extending the match to include all the dashed lines. SEARCHFORPROGRESS then fails to make any further progress, as no rules are applicable to the unmatched portions of the graphs.

Table 1. Pseudo-code for PA, a precedent analysis algorithm.

```
Inputs:    PRECEDENT consists of a high-level graph, a low-level
             graph, and their variable correspondences.
           DG is a set of design grammar rules.
Outputs:   One or two grammar rules.
Variables: LLG and HLG are graphs.
           PM is a partial match between two graphs.
           PROG? is a flag stating whether progress was found.

Procedure PA(PRECEDENT, DG)

  Let PM be the variable correspondences for PRECEDENT.
  Let LLG be the low-level graph for PRECEDENT.
  Let HLG be the high-level graph for PRECEDENT.
  Repeat
    Let [PROG?, HLG, PM] be SearchForProgress(LLG, HLG, PM, DG).
  Until PROG? is False.
  Return MakeConjectures(LLG, HLG, PM).
```

The overall criterion for extending the partial match is that matched nodes should represent functionally equivalent connection points or blocks. If one node is constrained relative to the inputs or outputs, the other should be identically constrained. The partial match is extended incrementally inward from the input variables and output variables according to two heuristics.

1. Two connection points (one from each graph) may be matched when their values are determined by the same function of corresponding matched nodes. (This moves in from the inputs.)
2. Two connection points may be matched if all of the following conditions hold: (a) They are inputs to the same type of functional block; (b) the blocks to which they are input drive matched nodes; and (c) there is no ambiguity in matching the other inputs to the functional blocks. There is ambiguity if either block has more than one input of the same type, and at least two of those inputs remain unmatched. (This moves in from the outputs.)

To illustrate these criteria, consider the pair of graphs in Figure 2. The first criterion would say that, since connection points α and α' are each XOR of corresponding previously matched nodes, they (and the XOR blocks) may be matched. The second criterion would say that, since β and β' are each the unambiguous inputs of the Z^{-1} boxes that drive y, they may be matched. The second criterion would be used again to extend the match to that shown in Figure 4.

There are some subtleties: consider what happens to the first criterion when *one graph* has two nodes that are (syntactically) the same function of the same inputs. There is no syntactic way of choosing between them. PA solves this using the fact that the graphs represent pure functions. The meaning of the graph does not change if we remove one of the two and replace its connections

Table 2. Pseudo-code for SearchForProgress, a subroutine of PA.

```
Inputs:    LLG and HLG are graphs.
           PM is a partial match between the two graphs.
           DG is a set of design grammar rules.
Outputs:   A flag stating whether progress was found.
           A modified high-level graph.
           An extended partial match between the two graphs.
Variables: BEST.GRAPH and QHLG are modified high-level graphs.
           BEST.PARTIAL.MATCH and QPM are extended partial matches.
           QUEUE is a queue of graphs, matches, and their depths.
           QDEPTH is the current depth of the search.
           SearchDepth is the allowed depth of search.

Procedure SearchForProgress(LLG, HLG, PM, DG)

  Let [BEST.GRAPH, BEST.PARTIAL.MATCH] be [HLG, PM].
  Let QUEUE be <[HLG, PM, 0]>.
  While QUEUE is not empty,
    Let [QHLG, QPM, QDEPTH] be Head(QUEUE).
    Let QUEUE be Tail(QUEUE).
    If CanExtendMatch?(LLG, QHLG, QPM),
       Then return [True, QHLG, ExtendMatch(LLG, QHLG, PM)].
    If size(QHLG) is less than size(BEST.GRAPH),
       Then let [BEST.GRAPH, BEST.PARTIAL.MATCH] be [QHLG, QPM].
    If QDEPTH is less than SearchDepth,
       Then add ApplyAllowableDesignRules(QHLG, QPM, QDEPTH, DG)
                to the back of QUEUE.
  Return [False, BEST.GRAPH, BEST.PARTIAL.MATCH].
```

by connections to the other. A straightforward pre-pass over a graph merges the syntactic equivalence classes into one node each.

There is another criterion for extending the partial match when a subgraph transforms into a single connection point. (This happens when a subgraph is functionally equivalent to the identity function.) In such cases, the previous two criteria do not apply, as there are no new nodes to which to extend the match. In this case, the algorithm looks for a situation in which the inverse image of a connection point under the partial match decreases. For example, the system would judge progress after transformation of "y =BUFFER(a)" to "a." That is, originally the partial match mapped both y and a to the same connection point, x, in the other graph. After the transformation, the only connection point mapped to x is a; thus the inverse image of x under the partial match has decreased from $\{a, y\}$ to $\{a\}$.

Once the partial match has been extended as much as possible, it is straightforward to construct the two rules. In the above example, the unmatched subgraphs consist of those functional blocks with no dashed line correspondence. In this case, PA creates a new functional block (f) to be the LHS of the new rules (as shown in Figure 5). Note that the inputs and outputs of the new block

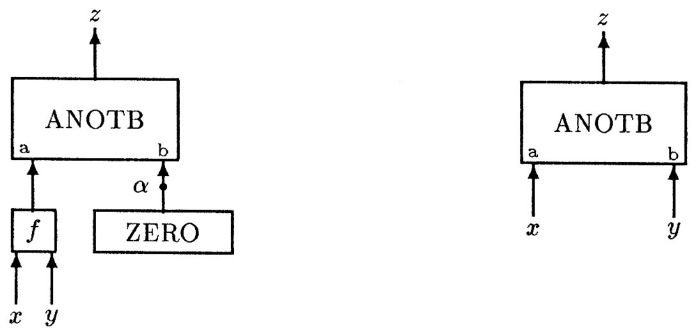

Figure 6. A precedent on which the second match-extension heuristic fails. ANOTB represents the function AND(a, NOT(b)). The heuristic would match connection point α with y, because there is no ambiguity in matching the inputs to the ANOTB boxes and they both drive z. This leads to a highly context-dependent and rather useless rule conjecture.

correspond to the connection points impinging on the unmatched subgraphs. Their correspondence is given by the partial match.

2.4.3 Correctness of the generated rules

At first glance, it might seem that PA can generate "incorrect" rule conjectures, since the heuristics for match extension can erroneously associate two nodes that really do not correspond functionally. For example, in Figure 6, the second match-extension heuristic would cause the system to associate node α in the left graph with node y in the right graph, because the inputs to the ANOTB function are unambiguous. The system would then conjecture the synthesis rule saying "sometimes replace constant ZERO with a connection to y and simultaneously replace $f(x,y)$ with x." This rule is clearly undesirable in that it can rarely be applied (at a minimum, y must be connected to ZERO). A designer would not really use this rule, because it is very specific and context-dependent.

However, this rule is not *incorrect*, since it can be applied in at least this one case. The same goes for any rule conjectured by PA.[7] Furthermore, since rule applications are checked analytically before they are executed, such rules cannot lead to functionally incorrect designs. However, they can lead to some rather strange design derivations, and they tend to be rather useless in general. I have not explored how often these faulty rules are conjectured ("how often" is a difficult question to address in the absence of real-world data), nor have I explored whether they cause more bad rule conjectures when they are present than when absent. These are two interesting questions for future research.

[7]Throughout, I assume that the teacher only presents correct precedents. If the precedent were incorrect, it would be possible for PA to conjecture a rule that was never applicable, hence incorrect.

Table 3. Pseudo-code for PA-NO-RR, an algorithm that learns using precedent analysis without rule reanalysis.

```
Inputs:     PRECEDENT consists of a high-level graph, a low-level
              graph, and their variable correspondences.
            DESIGN.GRAMMAR is a set of design grammar rules.
Outputs:    A revised design grammar.
Variables:  CDG is the current design grammar.
            T is a pair of learned design rules.

Procedure PA-NO-RR(PRECEDENT, DESIGN.GRAMMAR)

  Let CDG be DESIGN.GRAMMAR.
  Let T be PA(PRECEDENT, CDG).
  If T does not duplicate information in CDG,
     Then let CDG be the union of T with CDG.
  Return CDG.
```

2.5 Rule reanalysis

In addition to the issue of the conjectures' context-dependence, their specificity is also important. Precedent analysis may produce overly specific rules for two main reasons. First, since the matching heuristic moves in from the "edges" of the graph, it can get stuck early if some key unknown transformation applies near the inputs or outputs. Second, there might be more than one unknown rule used in constructing the precedent. In either case, the learned rules will have RHSs that are combinations of more than one unknown, more general rule. The learner is much less likely to see again a given complex combination of rule instances than it is to see such instances used separately in other contexts.

However, it is possible to learn new rules later that would let precedent analysis find the more general rules from which the first was composed. The idea of rule reanalysis is that whenever the learner induces a new rule, it should try to use the rule to analyze previously learned rules as though they were precedents. Winston (1975) introduced the idea of the near-miss felicity condition as a means of ensuring that no more than one new condition was used in any precedent. Rule reanalysis can be seen as an attempt at dealing with far misses by keeping overly-specific intermediate results around for later generalization.

This subsection demonstrates that reanalyzing existing rules leads to a more powerful learning method (in a sense to be defined) than one that does not. It also shows that rule reanalysis is still somewhat sensitive to the order of presentation of the precedents.

2.5.1 The need for reanalysis

For comparison purposes, Table 3 presents a pseudo-code description of PA-NO-RR, a simple program for accepting precedents, using PA to conjecture new rules, and adding the conjectures to the design grammar. This algorithm does *not* use rule reanalysis. Suppose one always presents precedents to PA-NO-RR in the best possible order, i.e., the order that results in the most general learned rules. Might it not be that rule reanalysis is a waste of time? The answer to this is "no," as the following counterexample shows.

Suppose that, unknown as yet to the system, there are four general design rules represented in three precedents. The four design rules are as follows, for brevity stated as propositions rather than graphically:

- $f_1(x) \iff g_1(x)$
- $f_2(x) \iff g_2(x)$
- $f_3(x,y) \iff g_3(x,y)$
- $f_4(x) \iff g_4(x)$

The three precedents are the following:

- $f_3(f_1(x), f_2(y)) \equiv g_3(g_1(x), g_2(y))$
- $f_2(f_4(t)) \equiv g_2(g_4(t))$
- $[f_3(f_1(z), f_2(z)), z = f_4(w)] \equiv [g_3(g_1(z'), g_2(z')), z' = g_4(w)]$

Further suppose that the system starts with an empty design grammar.

Here is the new design grammar that results from applying PA-NO-RR to the three precedents in the given order:[8]

- $b_1(x,y) \iff f_3(f_1(x), f_2(y))$
- $b_1(x,y) \iff g_3(g_1(x), g_2(y))$
- $b_2(t) \iff f_2(f_4(t))$
- $b_2(t) \iff g_2(g_4(t))$
- $f_4(x) \iff g_4(x)$

In this case, b_1 and b_2 are (arbitrary) names for the new blocks. Thus, only one of the four original design rules was discovered. In other words, simple precedent analysis is not sufficient to find the desired rules even when it sees precedents in the optimal order.

2.5.2 An example of rule reanalysis

Now let us see how PA-RR, a rule reanalysis algorithm, behaves on the same inputs. Table 4 presents a description of this algorithm, which (unlike PA-NO-RR) keeps reanalyzing old rules until no new conjectures emerge.

Suppose that PA-RR is presented with the same precedents above and in the same order. Since PA can analyze neither precedent 1 nor 2, the system conjectures four rules, one rule for each graph of each precedent: $b_1(x,y) \iff f_3(f_1(x), f_2(y))$, $b_1(x,y) \iff g_3(g_1(x), g_2(y))$, $b_2(t) \iff f_2(f_4(t))$, and $b_2(t) \iff g_2(g_4(t))$.

On seeing the third precedent, PA analyzes it using the b_1 rules. This results in the new rule $f_4(x) \iff g_4(x)$, and PA-RR then reanalyzes the

[8]Depending on the implementation details of PA, it could produce a slightly different partial parse, yielding the two rules, $h(z,w) \iff f_3(f_1(z), w)$ and $h(z,w) \iff g_3(g_1(z), w)$, in place of the f_4 rule shown.

Table 4. Pseudo-code for PA-RR, an algorithm that reanalyzes old rules using newly conjectured rules.

```
Inputs:    PRECEDENT consists of a high-level graph, a low-level
             graph, and their variable correspondences.
           DESIGN.GRAMMAR is a set of design grammar rules.
Outputs:   A revised design grammar.
Variables: CDG is the current design grammar.
           QUEUE is a queue of design grammar rules.
           T is a pair of learned design rules.

Procedure PA-RR(PRECEDENT, DESIGN.GRAMMAR)

  Let CDG be DESIGN.GRAMMAR.
  Let QUEUE be <PRECEDENT>.
  While QUEUE is not empty,
    Let T be PA(Head(QUEUE), Difference(CDG, Head(QUEUE))).
    If T does not duplicate information in CDG,
       Then let QUEUE be all rules in CDG.
            Let CDG be the union of T and CDG.
       Else let QUEUE be Tail(QUEUE).
  Return CDG.
```

existing rules. Using its new knowledge, PA can now analyze the b_2 rules to obtain another new rule, $f_2(x) \iff g_2(x)$. PA-RR again reanalyzes the existing rules, and applying PA to one of the b_1 rules results in the two simpler rules: $h(z,w) \iff f_3(f_1(x))$ and $h(z,w) \iff g_3(g_1(x))$. In summary, PA-RR has learned the following rules.[9]

- $f_4(x) \iff g_4(x)$,
- $f_2(x) \iff g_2(x)$,
- $h(z,w) \iff f_3(f_1(z),w)$,
- $h(z,w) \iff g_3(g_1(z),w)$.

To compare the performance of PA-NO-RR and PA-RR, we can define a generality partial order on design grammar rule sets by set inclusion between the derivations generated by the rule sets. Clearly, the set produced by PA-RR suffices to derive all the rules produced by PA-NO-RR. However, there is no derivation of the f_2 rule in terms of the rules produced by PA-NO-RR. Thus, PA-RR results in a (strictly) more general set of rules. In fact, all six orders of presentation result in strictly less general sets of rules when PA-NO-RR is used than when PA-RR is used.

This example shows that without rule reanalysis, the system can require *more* precedents to reach a given level of generality. There is clearly a tradeoff between the cost of using rule reanalysis and the cost of obtaining more precedents; however, this tradeoff is complex and heavily dependent on the implementation. We can observe two clear cases when rule reanalysis would

[9]The system still retains the b_i rules, even though they can be derived from the others.

be desirable. The first occurs when it is crucial that the learner get as much as possible from each precedent. For instance, if very few precedents are available, PA-NO-RR may not be able to formulate general rules. The second case occurs when the learner only receives precedents very slowly relative to its computing speed. This could easily happen in learning apprentice tasks, where the precedents are entered by a human.

2.5.3 Order sensitivity

PA-RR is somewhat sensitive to the order of presentation of precedents. This is because the algorithm does not find all maximal partial parses of all precedents every time it conjectures a new rule. The system saves a significant amount of processing by keeping only the conjectured new portion of the precedent. The alternative is keeping the entire precedent as well. This would at least double the amount of processing done by PA-RR.

To illustrate the order sensitivity, consider a slightly different set of precedents:

1′. $f_1(f_4(x)) \equiv g_1(g_4(x))$
2. $f_2(f_4(t)) \equiv g_2(g_4(t))$
3. $[f_3(f_1(z), f_2(z)), z = f_4(w)] \equiv [g_3(g_1(z'), g_2(z')), z' = g_4(w)]$.

Two of these are the same as in the previous example, but precedent 1 has been replaced by precedent 1′.

If these are presented to PA-RR in the order (2, 3, 1′), then the system generates only two new rule conjectures (aside from those corresponding to precedents 1′ and 2 themselves):

$$h(z,w) \Longleftrightarrow f_3(f_1(z),w) \quad \text{and} \quad h(z,w) \Longleftrightarrow g_3(g_1(z),w).$$

On the other hand, if the precedents are presented to PA-RR in the order (1′, 3, 2), then it instead conjectures the new rules (again, aside from those corresponding to precedents 1′ and 2 themselves):

$$h'(z,w) \Longleftrightarrow f_3(z,f_2(w)) \quad \text{and} \quad h'(z,w) \Longleftrightarrow g_3(z,g_2(w)).$$

These are incomparable rule sets (in the generality partial order defined above), because there are no derivations of the h rules in terms of the second rule set, and there are no derivations of the h' rules in terms of the first rule set.

The difference arises because PA-RR "forgets" some of the information in precedent 3 when it uses PA to isolate the portion of the precedent it does not understand. For example, once it analyzes the third precedent in terms of precedent 1′ and discards the explained part of the former, precedent 2 cannot be used to analyze either the precedent 1′ rules or the h' rules (the two "halves" of precedent 3). This is because precedent 2's graphs only match subgraphs of precedent 3 that are partially contained both in precedent 1′ and the h' rules. PA-RR's prior knowledge (of precedent 1′) has effectively prejudiced it against learning the h rules from the combination of precedents 2 and 3. This behavior is reminiscent of the way a human's prior knowledge can sometimes hinder the acquisition of significantly new concepts.

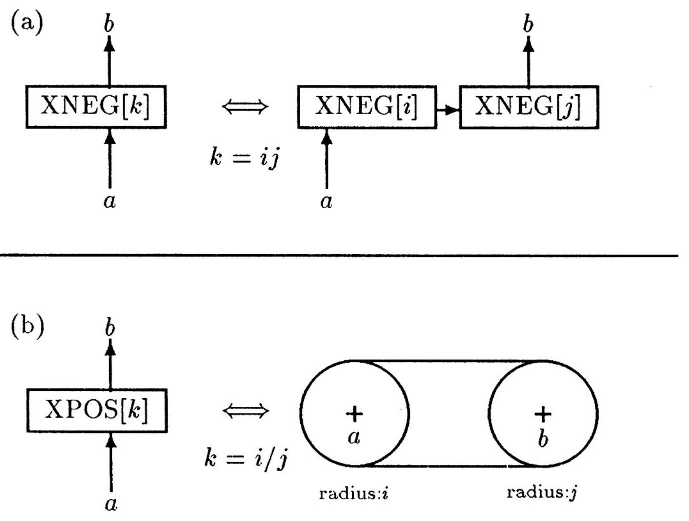

Figure 7. A small design grammar for the gear world. The blocks XPOS[k] and XNEG[k] represent multiplication of angular speed by $+k$ and $-k$, respectively. Note that these are parameterized rules. Rule (b) contains a pictorial representation of a sprocket and chain arrangement.

3. Evaluating the approach

Having described precedent analysis and rule reanalysis in some detail, we can now evaluate this approach to learning. This section examines the generality and limitations of learning by failing to explain. It also reports some preliminary experiments with the method and proposes some extensions suggested by these studies. The reader should keep in mind that the technique of learning by failing to explain is not intended as a complete model of learning. For learning tasks where complete grammar derivations of precedents are available to the learner (e.g., see Mooney & DeJong, 1985) an explanation-based approach is definitely more desirable for learning non-heuristic rules, since it produces new grammar rules that are both general and justified. If complete derivations are not available, either due to an incomplete grammar or to an inadequate explanatory mechanism, then an approach like that of PA-RR could be useful.

3.1 Generality of the method

Learning by failing to explain is at heart a method of grammar induction. As such, it should generalize to other domains in which knowledge can be formalized grammatically, at least in part. In addition to the domain of circuit design, the current system should work with only minor modification in any design domain in which knowledge may be represented as a grammatical hier-

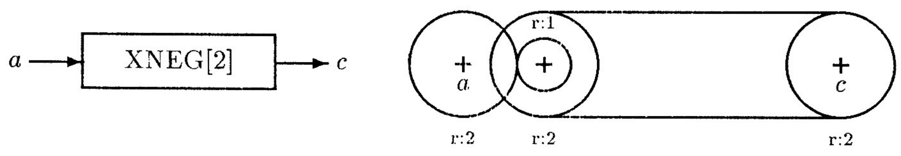

Figure 8. A gear-world precedent, depicting two abutting gears with radii of sizes two and one, together with a sprocket-chain-sprocket connection. Note that one sprocket shares a common shaft with the radius one gear. Shaft labels represent angular speed variables.

archy of function composition. Rich's (1981) plan calculus is one such grammar formalism for representing algorithmic knowledge about program design.

Though no experiments have been performed using the Plan Calculus, PA-RR-GW, a system related to PA-RR, has been applied to learning structural implementation rules for simplified gears. In this domain, functional constraints take the form of simple arithmetic relationships among shaft speeds.

As an example, suppose PA-RR-GW starts with the design grammar shown in Figure 7. The functional blocks XPOS[k] and XNEG[k] represent multiplication (of angular speed) by positive and negative constants, respectively. Note that these blocks are parameterized by the multiplicative constant. The first grammar rule expresses the fact that b = XNEG[ij](a) can be implemented by b = XPOS[i](XNEG[j](a)). The second rule expresses the idea that the sprocket-chain-sprocket connection implements the XPOS[k] function, with k equal to the ratio of the radii of the sprockets.

If PA-RR-GW is then shown the precedent in Figure 8, its precedent analysis component explains the sprocket portion using the grammar rules. The remainder forms the conjecture shown in Figure 9. Note that this conjectured rule is not general, in that its parameters are numbers rather than symbols. PA-RR-GW waits until it finds other instances of rules using exactly two gears to implement the XNEG function, then hands the set of qualitatively similar rule instances to a constructive generalization routine. This routine induces the functional relationships that hold between the gears' parameters and XNEG's parameter.

As the example illustrates, learning by failing to explain is only partially applicable to the gear world: PA-RR-GW learns only the qualitative structural aspects of the implementation rules. Gear-world components have continuously varying parameters, so it is necessary to induce functional relationships among numbers in order to capture fully general rules. Since this task moves beyond grammar induction, PA-RR-GW requires an auxiliary routine.

The system has successfully generalized the rules governing transfer ratios of meshed gears and the rules governing sprocket-chain connections. The gross structural relationships, e.g., that a sprocket-chain-sprocket connection implements positive multiplication of angular speed, were conjectured using precedent analysis and rule reanalysis. The arithmetic relationships, such as the

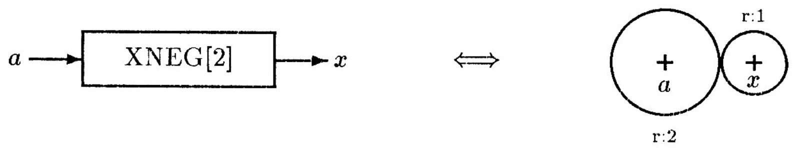

Figure 9. A rule conjectured by PA-RR-GW. Note that no numerical generalization has been done yet; further structurally isomorphic examples are required to allow the constructive generalization routine to operate. Only one rule was conjectured because the equivalence had one graph with only a single block.

equality of the multiplicative factor to the quotient of the sprocket radii, were generalized by the auxiliary routine. The details of the auxiliary routine are less important than its successful use in concert with techniques for learning by failing to explain.

Although the implementations of PA-RR and PA-RR-GW are somewhat dependent on the semantics of the graphs, the idea of maximal partial parsing should extend to any domain that can be formalized in grammar terms. In fact, similar work has been done in the domains of natural language syntax and simple arithmetic problems, as discussed in Section 4.

3.2 Limitations of the approach

There are two sorts of limitations on the PA-RR algorithm. The first affects the quality of the conjectured rules, whereas the second affects the computational efficiency of the system.

I observed two problems of the former kind. First, the second match-extension heuristic can fail in cases like those described in Section 2.4. This occurs when two inequivalent functional blocks have the same type, have matched outputs, and have uniquely corresponding unmatched inputs. Second, when the unknown rules are needed near the outer edges of the precedent, little progress can be made. This results in virtually the entire precedent being conjectured as the "new rule," with little or no generalization. For example, a precedent might incorporate unknown implementations of input and output buffers, yet be otherwise completely understood. Ideally, we would like the system to conjecture the input and output buffer implementations as new rules, but since the system has no way of extending the partial match from the inside out, it would conjecture just the entire precedent.

A major limitation on PA's efficiency is that it cannot *focus attention* on small portions of an example. Suppose we wish to show that $f(g(x)) \equiv j(k(x))$, and we can do this because we have rules $f \Leftrightarrow h_1$, $h_1 \Leftrightarrow h_2$, ..., $h_m \Leftrightarrow j$, $g \Leftrightarrow i_1$, ..., $i_m \Leftrightarrow k$. In this case, the system would try all possible derivations of length $m - 1$ before it would try the derivation of length m that allows progress. However, note that there are 2^{m-1} nodes in this search tree. If PA could focus its attention on just the f derivation by considering only rules pertaining to f and its derivations, it would only need to look through $m - 1$ nodes. This combinatorial interaction of multiple progress paths could be avoided by concentrating on one section of the graph at a time. On the other hand, Hall (1986a) shows that a uniformly depth-first approach is much worse than a uniformly breadth-first approach, because the system can waste time finding much longer derivations than necessary.

3.3 Illustrative experiments

I have not carried out experiments with "real-world" design data, as might be obtained from a human expert. However, Hall (1985) documents several examples to which the system has been successfully applied, and this section presents some empirical studies of PA's behavior.

In discussing the details of a PA run, it is useful to define the notion of a *progress history*. This is simply a list containing the numbers of derivation steps between successive discoveries of progress. For example, the progress history (2, 6, 2, (5)) indicates that in a fifteen-step partial derivation, PA found progress after the second, eighth, and tenth steps. The last lookahead phase failed to find progress, but a five step derivation resulted in the smallest graph (i.e., the best graph returned by SEARCHFORPROGRESS was a graph at level five of the lookahead tree).

In discussing the following experiments, it will be useful to define a base design grammar, G_1, which is shown in Table 5. Each rule includes a name, a left-hand side (LHS), and a right-hand side (RHS). The LHS takes the form OUTPUTS = BLOCK-NAME(INPUTS), where there may be multiple inputs and multiple outputs. The RHS specifies relations between the input and output variables that implement the functional specification in the left-hand side.

3.3.1 The importance of search depth

One factor that governs the success of PA is the value of SEARCHDEPTH, the maximum length of derivations that SEARCHFORPROGRESS will examine. There is a tradeoff between explanatory power and the amount of lookahead. Search time is (roughly) exponential in SEARCHDEPTH, since the lookahead tree contains roughly $b^{\text{SearchDepth}}$ nodes, where b is the average branching factor – the number of allowable rule applications at each level. Whenever the final lookahead phase is unsuccessful, the system must examine the entire tree. On the other hand, if the system uses a small SEARCHDEPTH, it may fail to explain an otherwise explicable precedent. Thus, it will generate a rule conjecture that can actually be derived from the preexisting grammar.

To examine the effect of search depth, I performed a simple experiment using the grammar G_1 and the precedent

$$\begin{gathered}[y = \text{NAND}(w, \text{AND}(w, \text{NAND}(c,d))), w = \text{NAND}(a,b)]\\ \equiv \\ [y = \text{OR}(w', \text{OR}(w', \text{AND}(c,d))), w' = \text{AND}(a,b)].\end{gathered}$$

Table 6 reports the results. With SEARCHDEPTH set to 2, PA conjectured the new rule $[r = \text{NOT}(\text{NOR}(p,q))] \equiv [r = \text{OR}(p,q)]$. In contrast, setting

Table 5. A design grammar for the circuit domain.

ADD2	LHS:	[s0, s1, co] = ADD2(ci, a0, b0, a1, b1)	
	RHS:	[s0, c] = ADD1(ci, a0, b0)	; a0 and b0, a1 and b1 commute
		[s1, co] = ADD1(c, a1, b1)	; c is an internal variable
ADD1	LHS:	[s, co] = ADD1(ci, a, b)	; a and b commute
	RHS:	s = XOR(ci, XOR(a, b))	
		co = OR(OR(AND(a, b), AND(a, ci)), AND(b, ci))	
XOR	LHS:	y = XOR(a, b)	; a and b commute
	RHS:	y = MUX(a, b, NOT(b))	
MUX	LHS:	y = MUX(s, a, b)	
	RHS:	y = OR(AND(s, b), AND(a, NOT(s)))	
NAND$_1$	LHS:	y = NAND(a, b)	; a and b commute
	RHS:	y = NOT(AND(a, b))	
NAND$_2$	LHS:	y = NAND(a, b)	; a and b commute
	RHS:	y = OR(NOT(a), NOT(b))	
NOR$_1$	LHS:	y = NOR(a, b)	; a and b commute
	RHS:	y = NOT(OR(a, b))	
NOR$_2$	LHS:	y = NOR(a, b)	; a and b commute
	RHS:	y = AND(NOT(a), NOT(b))	
ONE$_1$	LHS:	y = ONE( )	
	RHS:	y = OR(ONE, c)	; c is an internal variable
ONE$_2$	LHS:	y = ONE( )	
	RHS:	y = NOT(ZERO)	
ONE$_3$	LHS:	y = ONE( )	
	RHS:	y = OR(c, NOT(c))	; c is an internal variable
ZERO$_1$	LHS:	y = ZERO( )	
	RHS:	y = AND(ZERO, c)	; c is an internal variable
ZERO$_2$	LHS:	y = ZERO( )	
	RHS:	y = NOT(ONE)	
ZERO$_3$	LHS:	y = ZERO( )	
	RHS:	y = AND(c, NOT(c))	; c is an internal variable
BUFFER$_1$	LHS:	y = BUFFER(a)	
	RHS:	y = AND(ONE, a)	
BUFFER$_2$	LHS:	y = BUFFER(a)	
	RHS:	y = OR(ZERO, a)	
BUFFER$_3$	LHS:	y = BUFFER(a)	
	RHS:	y = a	; a single connection point
BUFFER$_4$	LHS:	y = BUFFER(a)	
	RHS:	y = NOT(NOT(a))	

Table 6. The effect of search depth on explanatory ability.

	FIRST CASE	SECOND CASE
INITIAL GRAMMAR SEARCH DEPTH	G_1 2	G_1 3
PROGRESS HISTORY CONJECTURED EQUIVALENCE	(1, 1, 2, (2)) $[r = \text{NOT}(\text{NOR}(p,q))]$ $\equiv [r = \text{OR}(p,q)]$	(1, 1, 2, 3, 2) PRECEDENT EXPLAINED

SEARCHDEPTH to 3 let the system explain the precedent using its current design knowledge, so no conjecture was necessary.

3.3.2 The effect of rule overuse

Another factor that influences PA's behavior is its current design grammar. Too many rules can cause needless search. On the other hand, the explanation for some precedents may require certain rules that are often applicable. For example, consider BUFFER Rule 3 in G_1. Each connection point can be expanded into a buffer, and the buffer can be re-implemented, say, as a double negation using BUFFER Rule 4. This sequence is allowable at every connection point of every graph. The system must have this rule available for some derivations, but it should not simply try it at every possible point. The maximum tolerable SEARCHDEPTH for a given rule base would be greatly reduced if such rules were used indiscriminately.

To study the effect of such rule overuse, I ran a second experiment using the precedent

$$[y = \text{XOR}(a, \text{ZERO})] \equiv [y = a].$$

and holding the search depth constant at eight. One condition used grammar G_1 as shown in Table 5. I stopped this run after two hours and hundreds of searched nodes with no progress found. The search had only reached level six of the tree on the first lookahead phase.

The second condition used a modified grammar, G_2, which disallowed certain directions of use for particular rules. This grammar was obtained from G_1 by disallowing all ZERO and ONE rule applications in the left to right direction (i.e., never allowing expansion of a ZERO block or a ONE block) and by disallowing right to left uses of the ADD1 rule. In this run, PA explained the same precedent with little search, expanding only 42 nodes. Table 7 summarizes the results of this experiment.

3.3.3 The use of derived rules

Another key insight demonstrated by the implementation is that *derived rules* can greatly increase explanatory power by decreasing the effective SEARCHDEPTH needed to find progress. Derived rules are just rules in the grammar that one knows can be derived from other grammar rules. The notion of a de-

Table 7. The detrimental effect of rule overuse on explanatory ability.

	FIRST CASE	SECOND CASE
INITIAL GRAMMAR	G_1	G_2
SEARCH DEPTH	8	8
NODES SEARCHED	> 200	42
PROGRESS HISTORY	—	(8)
CONJECTURED EQUIVALENCE	NONE (SYSTEM FAILED)	PRECEDENT EXPLAINED

rived rule is essentially the same as that of a chunked rule (Laird, Rosenbloom, & Newell, 1984) or a macrop (Fikes et al., 1972). The insight is that having a derivation of n steps summarized into a single step decreases the length of that derivation; hence, it increases the effective lookahead. This should let the system explain hopelessly complex precedents once it has first explained simpler precedents and stored the results as derived rules.[10]

To examine this effect, a third experiment was performed using the two-bit incrementer precedent:

$$\begin{gathered}[[s0, s1, co] = \mathrm{ADD2}(\mathrm{ZERO}, \mathrm{ONE}, a, \mathrm{ZERO}, b)] \\ \equiv \\ [co = \mathrm{AND}(a, b), s0 = \mathrm{NOT}(a), s1 = \mathrm{XOR}(a, b)].\end{gathered}$$

In one condition, PA was run on this precedent using a search depth of six and using grammar G_2. In this case, the system failed to explain the precedent. Analysis reveals that a search depth of eight would give a complete 41-step explanation, but this would involve expanding more than 10,000 nodes and require more than 12 CPU hours. The size of the search space can be largely attributed to the focus of attention problem mentioned previously.

In the second condition, PA was given the following series of four precedents:

- $\mathrm{XOR}(\mathrm{ZERO}, a) \equiv a$
- $\mathrm{XOR}(\mathrm{ONE}, a) \equiv \mathrm{NOT}(a)$
- $\mathrm{MUX}(\mathrm{ONE}, x, y) \equiv y$
- $[[co, s] = \mathrm{ADD1}(x, \mathrm{ONE}, \mathrm{ZERO})] \equiv [co = x, s = \mathrm{NOT}(x)]$

The fourth precedent required searching 123 nodes; its progress history was (2, 2, 8). After each one was fully explained, it was added to the grammar as a pair of rules (giving the new grammar G_3) and used in subsequent derivations. The system was then given the two-bit incrementer precedent, using grammar G_3 and a search depth of six. In this case, PA found a complete explanation for the precedent. Table 8 shows the results of this study.

[10]PA-RR and PA-RR-GW do not reanalyze derived rules, since the systems would successfully explain them. This detail was omitted earlier for the sake of brevity.

Table 8. The beneficial effect of derived rules on explanatory ability.

	FIRST CASE	SECOND CASE
INITIAL GRAMMAR	G_2	G_3
SEARCH DEPTH	6	6
NODES SEARCHED	> 10000	19
PROGRESS HISTORY	—	(2, 1, 1, 6)
CONJECTURED EQUIVALENCE	NONE (SYSTEM FAILED)	PRECEDENT EXPLAINED

To see how derived rules can improve learning, consider a related precedent, a two-bit incrementer with delayed carry output:[11]

$$\begin{gathered}[coD = \mathrm{Z}^{-1}(c), [s0, s1, c] = \mathrm{ADD2}(\mathrm{ZERO}, \mathrm{ONE}, a, \mathrm{ZERO}, b)] \\ \equiv \\ [coD = \mathrm{NOT}(\mathrm{PG}(\mathrm{NOT}(\mathrm{PG}(c', clk1)), clk2)), \\ c' = \mathrm{AND}(a, b), s0 = \mathrm{NOT}(a), s1 = \mathrm{XOR}(a, b)].\end{gathered}$$

Using the grammar G_2, the system could not find the desired rule conjecture,

$$[y = \mathrm{Z}^{-1}(x)] \Longleftrightarrow [y = \mathrm{NOT}(\mathrm{PG}(\mathrm{NOT}(\mathrm{PG}(x, clk1)), clk2))].$$

If SEARCHDEPTH were large enough to allow the required partial derivation, the run would simply take too long. On the other hand, PA does find this conjecture using grammar G_3 in a reasonable amount of time.

This study demonstrates another way in which PA-RR can be combined with another learning method – chunking – so the two together can learn a rule that neither can learn alone. The new rule cannot be learned from grammar G_2 by chunking, because the rule cannot be derived from G_2.

However, note that having extra grammar rules increases the "bushiness" of the search tree and, in principle, this could lead to worse performance. Minton (1985) has addressed this problem in his MORRIS system. The key insight there was to save only rules that are either frequently useful or useful in solving difficult problems. I have not attempted to implement these ideas within the PA system.

3.3.4 Additional tests

PA has been tested on several grammar-precedent pairs besides those already discussed. Most runs used the grammar G_1, shown in Table 5, or closely related grammars. These cases were intended to demonstrate both learning (when explanation fails) and successful explanation (hence no learning). The most complex precedent fully explained (requiring the longest derivation) in

[11]The PG block denotes a pass gate: a device that outputs its first argument when its second argument is one. Otherwise it "turns off," putting out high impedance. *Clk1* and *clk2* are inputs from the system's two-phase clock.

the tests to date is the twelve-step derivation of the one-bit incrementer. (This was the fourth precedent in the third experiment, as reported in Section 3.3.3.)

During runs in which the system learned a rule, the most complex case was the two-bit incrementer with delayed carry (the last example of Section 3.3.3). In this case, the progress history was (2, 5, (2)), a nine-step derivation. However, with SEARCHDEPTH = 7, the system examined all derivations of the form (2, 5, (k)), with $k < 7$, choosing the particular two-step derivation because it resulted in the smallest unmatched subgraph. PA conjectured the Z^{-1} implementation in terms of PG blocks, a much simpler rule than the entire precedent.

In all, I performed some 20 tests of the PA algorithm. PA-RR has not been tested as extensively. In one test, PA-RR started from an empty grammar and learned implementation rules for NOT, AND, and OR gates (in terms of transistor primitives). It inferred these rules from three precedents, each of which employed from three to six logic gates. In other tests, PA-RR started from some pre-existing grammar and was given a rule that allowed it to re-analyze some preexisting rule in the grammar.

3.4 Directions for future research

The research reported here has left many questions unanswered. One major open question addresses the nature and use of analytic knowledge needed for both learning and design. In particular, the system must be able to verify the allowability of rule applications. PA-RR cannot be fully automatic (nor can any design system) until this question has been addressed.

More work needs to be done on heterogeneous learning systems that combine many different learning techniques. This paper has shown how learning by failing to explain may be used in concert with numerical function generalization and chunking. A reasonable next step would be attempting to incorporate explanation-based learning for generalizing explicable precedents.

In order to unify and generalize the knowledge base further, the system must recognize when the function of a learned implementation is equivalent to one already known, so that the same LHS block name may be used for the new rule. This would increase the system's power to find derivations.

Further experimentation should also be done, especially using realistic sets of precedents. These might be obtained from human experts or teachers of design. This would be particularly interesting for experiments with a heterogeneous learning system, as the different learning techniques could be evaluated for usefulness and relative power.

Another area of future work lies in the improvement of PA-RR, the current implementation. The above experiments suggest several modifications:

- PA should not just try all rules uniformly at every lookahead step. It should have heuristics for avoiding, except as a last resort, certain classes of rule applications (like forward uses of ZERO and ONE rules as mentioned in Section 3.3). This would decrease the bushiness of the search tree and therefore increase the tolerable value of SEARCHDEPTH.

- PA-RR needs a MORRIS-like method (Minton, 1985) of forgetting some rules to cut down the bushiness of the search tree. For example, once a rule can be completely derived from other rules, it is a good candidate for pruning from the rule base (however, see the next item).
- PA should use derived (chunked) rules that appear commonly in derivations, as this increases the amount of lookahead per step. The increase in branching factor this implies must be traded off against the extra power gained. As Section 3.3 showed, adding the right chunks is well worth the slight increase in search space size.
- PA should have a mechanism to focus its attention on a given area of the example to cut down on the combinatorics of the lookahead tree.
- The performance of the match-extension heuristics might be improved through the use of analytic knowledge.

I believe that each of these additions would lead to a more robust system for learning by failing to explain.

4. Related work

In this section I consider some relations to earlier work that has employed similar learning methods. I have mentioned PA's relation to grammar induction, and some closely related research falls in this area. I also consider connections to work on integrated systems that incorporate both explanation-based and empirical learning components.

4.1 LPARSIFAL

Berwick's (1985) LPARSIFAL, a language-learning system, can be viewed as another system that learns by failing to explain. Of course, the grammars in the natural language domain use languages of strings instead of graphs. Also, Berwick's system is based on Marcus' (1980) PARSIFAL parser, a forward-chaining production system in which the rules can examine only the first three structures in the input buffer. But despite these differences, there are many similarities between the learning methods.

LPARSIFAL begins with a knowledge of $\overline{X}$ theory, which can be viewed as analytic knowledge for the natural language domain. The system first uses its current rules to parse an input sentence as much as possible. If the parse fails, the system tries to construct a single PARSIFAL rule that would let it complete the parse. If it fails to construct such a rule, the precedent is ignored. Otherwise, the new rule is added to the rule base. If the system acquires more than one rule with identical action parts and $\overline{X}$ contexts, it generalizes the condition parts by keeping only those structures common to the rules. This combination of learning by failing to explain with an inductive generalization technique is reminiscent of the way PA-RR-GW learns and generalizes structure rules in the gear world.

Berwick argues that LPARSIFAL can learn natural languages because they are naturally constrained. The action of ignoring precedents for which the system cannot construct a rule lets the system exploit the near-miss felicity condition without requiring a carefully crafted sequence of precedents. Thus, rule reanalysis is not necessary in this domain, since it is a method for handling far misses.

4.2 SIERRA

VanLehn's (1987) SIERRA system can also be viewed as a system that learns by failing to explain, though its author refers to this process as "learning by completing explanations." The domain is learning procedures for simple arithmetic calculations, and the system is intended as a psychological model of learning in young children. SIERRA learns from a sequence of "lessons," which are groups of worked subtraction problems. VanLehn describes his technique as attempting to explain the examples of the next lesson using the current procedure, then proposing a single new subprocedure that lets the system complete the derivations of the worked examples. This is analogous to precedent analysis.

The system depends on certain teacher-student felicity conditions to make this process tractable. These include the assumption that each new lesson introduces only one new subprocedure and the assumption that the teacher has shown all intermediate states of the computation for each example (e.g., making all borrows explicit). The first condition, one disjunct per lesson, is a variant of the near-miss idea. SIERRA assumes that the mysterious parts of the new lesson's precedents will be explained by only a single missing "rule." Using this condition again obviates the need for rule reanalysis.

VanLehn argues for a knowledge-acquisition methodology incorporating the above felicity conditions. There is obviously a tradeoff between ease for the teacher and ease for the student. Unconstrained precedent sets are easy for the teacher but difficult for the student. Felicity conditions make the student's job easier at the cost of requiring the teacher to know much more about the student's internal representations and knowledge states. Rule reanalysis is one way of mitigating this tradeoff, allowing the student to take a bit more of the responsibility, yet still not allowing the teacher arbitrary freedom. For example, the teacher would still need to exercise care that the example sequence did not allow too many different parses.

4.3 LAS

Anderson's (1977) LAS system[12] may also be viewed as learning by failing to explain. The system takes as input sentences paired with their meanings, which it represents in a semantic network. It attempts to construct an augmented transition network (ATN)[13] that would let it generate those sentences from their meanings, as well as construct the meanings from their associated sentences. Each time LAS sees a new sentence, it tries to extend its existing ATN so it can correctly parse that sentence. In other words, the system "ex-

[12]For a review of this system, as well as of LPARSIFAL, see Langley and Carbonell (1987).

[13]ATNs are basically context-free grammars with procedural semantics and augmentations that let them handle global properties like subject-verb agreement. Actually, LAS used a more limited class of grammars called *recursive* transition networks.

plains" as much of the sentence as possible with existing knowledge and then uses the remaining differences to slightly alter its grammar.

LAS can extend its ATN by adding new arcs, but it can also expand its existing word classes to allow a parse. For example, suppose the system has a grammar that recognizes the sentence, "The dog chased the ball." If one gives it the new input, "The dog chased the cat," it can see that the previous knowledge is sufficient to explain the new input if only the word class containing "ball" were expanded to include "cat." In addition, the program can merge two word classes when it finds they overlap sufficiently. Finally, LAS can merge sub-ATNs when they are similar enough. This approach achieves the effect of rule reanalysis, because it can store a far miss – i.e., a sentence that is quite dissimilar from any it has seen before – as a disjunctive branch of the ATN and later merge that branch with newer sub-ATNs.

LAS uses knowledge about concepts, like the corresponding word and relative significance, to assist generalization in a manner similar to explanation-based learning. It uses this knowledge to construct the initial ATN from the sentence itself. Thus, it can be viewed as heterogeneous in that it combines different learning techniques in one system.

4.4 OCCAM

To be robust, learning systems must incorporate many different kinds of techniques. I have argued that the combination of learning by failing to explain with chunking and numerical function induction is more powerful than any one in isolation. Similarly, adding explanation-based techniques would provide a powerful ability to prune irrelevant detail from precedents.[14]

Pazzani, Dyer, and Flowers' (1986) OCCAM system illustrates one approach to integrating explanation-based and empirical techniques. The system looks for correlations (empirical generalizations) in the input data and then attempts to justify the correlations using its causal theory. Empirical generalizations are based on a notion of similarity and a rule base containing heuristic generalization rules. When the justification fails, OCCAM postulates possible additions to the causal theory ("tentative generalizations"). These are assumed to hold until contradicted by later examples. When justification succeeds, the explanation is used to generalize the correlation further, via an explanation-based technique ("explanatory generalizations").

A major difference between OCCAM and PA-RR is that the former uses a fixed base of heuristics to form candidate rules from observations. In contrast, PA-RR uses current explanatory knowledge to help form a conjecture.

5. Conclusion

In this paper, I motivated the task of learning design grammar rules by showing that a design grammar facilitates four basic competences of design: top-down design, optimization, design derivation, and analogical design. A de-

[14]In this vein, Lebowitz (1986) also argues that systems should combine explanation-based and empirical techniques.

sign grammar represents the synthesis component of design knowledge, which is distinguished from theory-based analytic knowledge.

Precedent analysis is a method for learning from precedents that does not require the ability to explain a precedent completely before learning it. The key idea is that the learner uses current knowledge to partially understand a precedent before conjecturing a new rule, pruning away the portions of the precedent that are already understood. As with any inductive method, it can only produce rule conjectures. This idea was implemented in the PA system.

Rule reanalysis is a technique for using new rules to try to analyze old rules as if they were precedents. This approach can be viewed as a method for dealing with "far misses" by retaining overly specific rules until further learning allows their generalization. Even though rule reanalysis is somewhat sensitive to order of presentation of precedents, it is still inherently more powerful than simple acceptance of new rules, even assuming that the precedents are ordered optimally. The program PA-RR implements this idea; it accepts precedents, calls PA to analyze them, and then invokes rule reanalysis.

A related system, PA-RR-GW, has been implemented for learning design grammar rules in a simplified gear domain. This provides evidence for the generality of the techniques. Moreover, it shows how learning by failing to explain can be used together with another generalization technique, numerical function induction, to handle domains in which components have continuously varying parameters.

Several experiments were done to explore the properties of the system. Major influences on performance included the value of the SEARCHDEPTH parameter, the presence of frequently applicable (but ill-advised) rules, and the presence of chunked rules to decrease the minimum SEARCHDEPTH required. This last effect shows how the system can be profitably combined with yet another learning technique – chunking.

Acknowledgements

Thanks to Patrick Winston, for providing guidance on the original thesis. Thanks also to Tom Mitchell, Pat Langley, and the anonymous reviewers for many valuable comments on the paper. This research was supported in part by a National Science Foundation Graduate Fellowship and in part by the Advanced Research Projects Agency of the Department of Defense under Contract N00014-85-K-0124 from Office of Naval Research.

References

Anderson, J. R. (1977). Induction of augmented transition networks. *Cognitive Science*, *1*, 125–157.

Berwick, R. C. (1985). *The acquisition of syntactic knowledge.* Cambridge, MA: MIT Press.

Brotsky, D. C. (1984). *An algorithm for parsing flow graphs* (Technical Report Number AI-TR-704). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Darlington, J. (1981). An experimental program transformation and synthesis system. *Artificial Intelligence*, *16*, 1–46.

DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*, 145–176.

Ellman, T. (1985). Generalizing logic circuit designs by analyzing proofs of correctness. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 643–646). Los Angeles, CA: Morgan Kaufmann.

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

Hall, R. J. (1985). *On using analogy to learn design grammar rules.* Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Hall, R. J. (1986a). *Learning by failing to explain* (Technical Report AI-TR-906). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Hall, R. J. (1986b). Learning by failing to explain. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 568–572). Philadelphia, PA: Morgan Kaufmann.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1984). Towards chunking as a general learning mechanism. *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 188–192). Austin, TX: Morgan Kaufmann.

Langley, P., & Carbonell, J. G. (1987). Language acquisition and machine learning. In B. MacWhinney (Ed.), *Mechanisms of language acquisition.* Hillsdale, NJ: Lawrence Erlbaum.

Lebowitz, M. (1986). Not the path to perdition: The utility of similarity-based learning. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 533–537). Philadelphia, PA: Morgan Kaufmann.

Mahadevan, S. (1985). Verification-based learning: A generalization strategy for inferring problem-reduction methods. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 616–623). Los Angeles, CA: Morgan Kaufmann.

Marcus, M. (1980). *A theory of syntactic recognition for natural language.* Cambridge, MA: MIT Press.

Michalski, R. S., & Stepp, R. E. (1983). Learning from observation: Conceptual clustering. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Minton, S. (1985). Selectively generalizing plans for problem-solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 596–599). Los Angeles, CA: Morgan Kaufmann.

Minton, S., Carbonell, J. G., Etzioni, O., Knoblock, C. A., & Kuokka, D. R. (1987). Acquiring effective search control rules: Explanation-based learning in the PRODIGY system. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 122–133). Irvine, CA: Morgan Kaufmann.

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

Mitchell, T. M., Utgoff, P., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Mooney, R., & Bennett, S. W. (1986). A domain independent explanation-based generalizer. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 551–555). Philadelphia, PA: Morgan Kaufmann.

Mooney, R., & DeJong, G. (1985). Learning schemata for natural language processing. *Proceedings of the Ninth International Conference on Artificial Intelligence* (pp. 681–687). Los Angeles, CA: Morgan Kaufmann.

Pazzani, M., Dyer, M., & Flowers, M. (1986). The role of prior causal theories in generalization. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 545–550). Philadelphia, PA: Morgan Kaufmann.

Ressler, A. L. (1984). *A circuit grammar for operational amplifier design* (Technical Report AI-TR-807). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Rich, C. (1981). A formal representation for plans in the Programmer's Apprentice. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 1044–1052). Vancouver, B.C., Canada: Morgan Kaufmann.

Steinberg, L., & Mitchell, T. M. (1984). A knowledge based approach to VLSI CAD: The REDESIGN system. *Proceedings of the 21st IEEE Design Automation Conference* (pp. 412–418). Albuquerque, NM: IEEE.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*, 1–40.

Wills, L. M. (1987). *Automated program recognition* (Technical Report AI-TR-904). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision.* New York: McGraw-Hill.

Winston, P. H., Binford, T. O., Katz, B., & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 433–439). Washington, DC: Morgan Kaufmann.