Explanation-Based Learning: A Survey of Programs and Perspectives

THOMAS ELLMAN

Columbia University, Department of Computer Science, New York, New York 10027

Explanation-based learning (EBL) is a technique by which an intelligent system can learn by observing examples. EBL systems are characterized by the ability to create justified generalizations from single training instances. They are also distinguished by their reliance on background knowledge of the domain under study. Although EBL is usually viewed as a method for performing generalization, it can be viewed in other ways as well. In particular, EBL can be seen as a method that performs four different learning tasks: generalization, chunking, operationalization, and analogy.

This paper provides a general introduction to the field of explanation-based learning. Considerable emphasis is placed on showing how EBL combines the four learning tasks mentioned above. The paper begins with a presentation of an intuitive example of the EBL technique. Subsequently EBL is placed in its historical context and the relation between EBL and other areas of machine learning is described. The major part of this paper is a survey of selected EBL programs, which have been chosen to show how EBL manifests each of the four learning tasks. Attempts to formalize the EBL technique are also briefly discussed. The paper concludes with a discussion of the limitations of EBL and the major open questions in the field.

Categories and Subject Descriptors: I.2.6 [**Artificial Intelligence**]: Learning—*analogies*; *concept learning*; *induction*; *knowledge acquisition*

General Terms: Experimentation

Additional Key Words and Phrases: Analogy, analytic learning, chunking, concept acquisition, empirical learning, explanation-based learning, generalization, goal regression, machine learning, operationalization, similarity-based learning

INTRODUCTION

Research in the field of machine learning has identified two contrasting approaches to the problem of learning from examples. The traditional method is sometimes known as **empirical learning** or **similarity-based learning**.[1] This technique involves examining multiple examples of a concept in order to determine the features they have in common. Researchers using the empirical approach have assumed that an intelligent system can learn from examples without having much prior knowledge of the domain under study. Some well-known examples of empirical learning are given in Winston [1972], Michalski [1980], and Lebowitz [1983], among others. This research is surveyed by Angluin and Smith [1983], Cohen and Feigenbaum [1982], Michalski [1983], Michalski et al.

[1] A glossary of selected terms is given at the end of this paper. Each term in the glossary is printed in boldface the first time it appears in text.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0360-0300/89/0600-0163 $1.50

CONTENTS

[1983], and Mitchell [1982a]. An alternative technique known as **explanation-based learning** (EBL) has been developed more recently. This **analytic learning** method attempts to formulate a **generalization** after observing only a single example. In contrast to empirical learning techniques, EBL requires that a learning system be provided with a great deal of domain knowledge at the outset. Some examples of the EBL technique are given by Mitchell [1983], DeJong [1986], Carbonell [1986], and Mostow [1983a], among others described below.

I.1 An Intuitive Example of EBL

EBL is based on the hypothesis that an intelligent system can learn a general concept after observing only a single example. In order to illustrate how this can be done, consider the following example taken from the card game "Hearts."[2] Imagine a student who is learning to play the game of Hearts by looking over the shoulder of a teacher who is actually playing the game. The teacher is faced with the situation described in Figure 1. The leader of the current trick has just played the eight of hearts. According to the rules, the teacher must play one of his hearts. He can choose either the queen, the seven, the four, or the two of hearts. It turns out that the teacher chooses to play the seven of hearts. The student might explain the teacher's choice with the following line of reasoning:

(1) This trick contains hearts. The winner of the trick will accumulate some undesirable points. Therefore, it is best to play a card that will lose the trick.
(2) Playing a high card will minimize the chances of taking tricks in the future. All other things being equal, it is better to play a high card than a low card.
(3) The seven of hearts was chosen because it is the highest heart that is guaranteed to lose the trick.

After explaining the example, the student might realize that the same line of reasoning would also apply in slightly different situations. Although this example was taken from the fourth trick of the game and the players all had specific scores, these facts were not used in the explanation. Furthermore, the explanation does not depend on the ranks of any cards in the teacher's hand, other than hearts. The explanation would continue to be valid even

[2] Hearts is normally played with four players. Each player is dealt 13 cards. At the start of the game, one player is designated to be the "leader." The game is divided into 13 successive tricks. At the start of each trick, the leader plays a card. Then the other players play cards in order, going clockwise around the circle. Each player must play a card matching the suit of the card played by the leader, if he has such a card in his hand. Otherwise, he may play any card. The player who plays the highest card in the same suit as the leader's card will take the trick and become the leader for the next trick. Each player receives one point for every card in the suit of hearts contained in a trick that he takes. In the simplest version of the game, the objective is to minimize the number of points in one's score. Other versions are more complicated. Complete rules are found in Gibson [1974].

Trick number:	4	
Current scores:	TEACHER:	0
	TOM:	0
	DICK:	2
	HARRY:	0
Lead suit:	♥	
Cards on table:	♥ 8	
Teacher's hand:	♠ JACK, 7	
	♥ QUEEN, 7, 4, 2	
	♦ ACE, 10	
	♣ JACK, 9	
Teacher's card choice:	♥ 7	

Figure 1. A training example from Hearts.

Trick number:	6	
Current scores:	STUDENT:	3
	TOM:	0
	DICK:	2
	HARRY:	0
Lead suit:	♠	
Cards on table:	♠ 10, ♥ QUEEN	
Student's hand:	♠ ACE, 8, 4	
	♥ JACK, 10	
	♦ 9, 5	
	♣ 4	
Student's card choice:	♠ 8	

Figure 2. A new Hearts example covered by the general rule.

if these features were changed. By eliminating such irrelevant facts, the student could formulate a general rule. The rule might say, "Whenever a trick contains hearts, play the highest legal card guaranteed to lose the trick." The explanation can help the student formulate this general rule. The rule could be created by separating the facts used in the explanation from the facts that are irrelevant to the explanation. Using this rule, the student could determine which card to play in new situations like the one described in Figure 2. In this situation the rule would recommend playing the eight of spades, because it is the highest spade guaranteed to lose the trick.

The term *explanation-based learning* has been used to encompass a wide variety of methods. Nevertheless, most of these methods can be understood in terms of the two-step procedure used by the student described above. The first step is to build an explanation of the function or behavior of the input example. The explanation is intended to capture a general principle of operation embodied in the example. In order to build the explanation, the system must be provided with some background knowledge of the domain. The second step involves analyzing the explanation and the example in order to construct a generalized concept. Features and constraints pertaining to the example are generalized as much as possible, as long as the explanation remains valid. The generalization will include other examples that can be understood using the same explanation and that manifest the same principle of operation. EBL is a method of using background knowledge to determine which features and constraints on an example can be generalized. The generalizations are justified, since they can be explained in terms of the system's background knowledge. For this reason one may speak of EBL as a type of **justified generalization**.

I.2 Overview

Section 1 contains a discussion of why EBL methods are necessary. Some of the issues and problems that EBL techniques are intended to address are described here. The history of EBL is also described, showing how it developed out of several different branches of the machine learning field. This section also describes the relation between EBL and other knowledge-intensive learning techniques.

Section 2 is a survey of some representative EBL programs, illustrating that EBL methods apply to a variety of learning tasks including generalization, chunking, operationalization, and analogical reasoning. It is divided into four parts corresponding to these four learning tasks. For each type of task, several EBL programs that perform the task are described. This section also shows that differences between the four categories of EBL programs are largely a matter of interpretation. The operation of most EBL programs can be interpreted in terms of any of the four learning tasks.

In Section 3, efforts to precisely define the methods of EBL, the requirements for building EBL systems, and the types of learning tasks that EBL can handle are described. Formalization also serves to clarify the relation between the four categories of EBL systems. Section 4 is an attempt to characterize the types of learning that EBL systems can and cannot perform. Section 5 contains a discussion of major open problems in the EBL field and some ongoing attempts to resolve them.

1. BACKGROUND OF EBL

1.1 Why Is EBL Necessary?

The methods of explanation-based learning have been developed to address several different issues in the field of machine learning. One issue involves human learning abilities. Some EBL research has been motivated by the observation that people are often able to learn a general rule or concept after observing a single instance of the concept. Experimental evidence for single-instance learning among humans is reported in Ahn et al. [1987]. Textbooks also provide some evidence for this type of learning. For instance, a textbook on logic circuits presents an example of a three-bit shift register and then asks the student to design a four-bit shift register as an exercise [Mano 1976, p. 78]. In order to solve the problem, the student must somehow generalize or transform the single example of a three-bit shift register. Empirical learning techniques are not suitable for learning from a single example. They normally require examining multiple instances of a concept. EBL is specifically designed for generalizing from a single example and is therefore able to model a type of human learning outside the scope of empirical methods.

EBL methods also address a more theoretical issue. EBL may be viewed as an attempt to solve the problem of inductive **bias**. As described by Mitchell [1980], every system that learns from examples requires some sort of bias. Mitchell defines bias to be "any basis for choosing one generalization over another, other than strict consistency with the observed training instances" [Mitchell 1980, p. 1]. A system lacking inductive bias would not be capable of making predictions beyond the training examples it has already seen. Typical types of bias include using a restricted vocabulary in the generalization language [Utgoff 1986] and preferring maximally specific concept descriptions [Dietterich and Michalski 1981] among others [Dietterich 1986]. EBL may be viewed as an attempt to use "background knowledge" or a "domain model" as a type of bias. The EBL method is biased toward making generalizations that can be justified by explaining them in terms of the domain model. EBL programs usually represent domain knowledge in a declarative style, and may therefore be said to utilize a declarative bias representation.

Several advantages result from representing bias in terms of a declarative domain model [Russell and Grosof 1987]. To begin with, a declarative bias can be interpreted in terms of direct statements about the domain. For this reason, the bias is subject to evaluation by human experts even before it is used to process training examples. In comparison, a nondeclarative bias such as a restricted vocabulary is not immediately interpretable as a statement about the domain [Dietterich 1986]. It therefore cannot be easily evaluated except by testing its consistency with the training examples. A declarative bias also offers advantages of domain independence. As observed by Dietterich and Michalski [1981], greater domain independence is achieved if the bias is contained in a separate module. The declarative domain models used by EBL systems are usually kept separate and can be easily modified. Traditional types of bias, such as the two cited above, are normally built into the representation and procedures used by the learning system. For this reason they are not easily modifiable. A declarative bias representation also helps to integrate diverse sources of background knowledge into the learning process [Russell and Grosof 1987].

1.2 The History of EBL

Explanation-based learning has only recently emerged as a recognizable area of study. Consequently, most early EBL research was undertaken by investigators who were not working on "explanation-based learning" per se. EBL may be viewed as a convergence of several distinct lines of research within machine learning. In particular, EBL has developed out of efforts to address each of the following problems:

- **Justified generalization**: A logically sound procedure for generalizing from examples. Given some initial background knowledge B and a set of training examples T, justified generalization finds a concept C that includes all the positive examples and excludes all the negative examples. The learned concept C must be a logical consequence of the background knowledge B and the training example set T [Russell 1986].
- **Chunking**: In the context of explanation-based learning, chunking is a process of compiling a linear or tree-structured sequence of operators into a single operator. The single operator has the same effect as the entire original sequence [Rosenbloom and Newell 1986].
- **Operationalization**: A process of translating a nonoperational expression into an operational one. The initial nonoperational expression may be a set of instructions or a concept. Concepts and instructions are considered to be operational with respect to an agent if they are expressed in terms of actions and data available to the agent [Mostow 1983a].
- **Justified analogy**: A logically sound procedure for reasoning by analogy. Given some initial background knowledge B, an analog example X, and a target example Y, find a feature F such that F(X) is true, and infer that F(Y) is true. The conclusion F(Y) must be a logical consequence of F(X) and the background knowledge B [Davies and Russell 1987].

Two of the first investigators to develop EBL methods were DeJong and Mitchell. DeJong's first paper in the EBL genre was DeJong [1981], in which he outlines a method of using explanations to learn procedural schemata from natural language input. DeJong viewed his approach as an attempt to model "insight learning" that involves "grasping a principle" embodied in an example [DeJong 1981, p. 67]. Mitchell's first EBL program was the LEX-II system developed jointly with Utgoff. This system involved a method of learning search control heuristics by analyzing sequences of operators [Utgoff and Mitchell 1982]. Mitchell's overall approach to EBL was first outlined in his "Computers and Thought" paper [Mitchell 1983], where he suggested that a learning system be given "declarative knowledge of its learning goal" [Mitchell 1983, p. 1145]. Such knowledge would enable a system to make "justifiable" generalizations and would be more powerful than purely "empirical" or "syntactic" methods.

At the same time that Mitchell and DeJong were developing EBL methods of generalization, Carbonell introduced his method of derivational analogy [Carbonell 1983a]. Carbonell's method uses derivations as a guide to analogical reasoning in a manner similar to the way in which EBL uses explanations to guide generalization. Winston was another one of the first investigators to use EBL methods in the context of reasoning by analogy [Winston et al. 1983]. The EBL methods used by Carbonell and Winston are both similar to Gentner's "structure-mapping" theory of analogy [Gentner 1983]. They also resemble Banerji and Ernst's method of using homomorphisms to implement a type of analogical reasoning [Banerji and Ernst 1972].

One of the first operationalizing systems was Mostow's FOO program for operationalizing advice [Mostow 1981]. Keller's LEXCOP technique [Keller 1983] was another early example of operationalization. The techniques used by Keller and Mostow bear a strong resemblance to Balzer's method of "transformational implementation" [Balzer et al. 1976]. A general approach to the problem of operationalizing advice is discussed in Hayes-Roth and Mostow [1981]. This line of research can be ultimately traced back to McCarthy's suggestion for an advice taking program [McCarthy 1968]. All of these systems may be seen as implementing a type of "learning by being told" [Cohen and Feigenbaum 1982].

Early research into chunking of operator sequences includes the STRIPS system [Fikes et al. 1972] along with Lewis [1978] and Neves and Anderson [1981]. Although STRIPS uses explanation-based methods for generalizing robot plans, it was not viewed as an EBL system by its authors, since it was built well before EBL became a recognized field of study. The idea of combining individual operators into macros goes back to Amarel's paper on representations for the "missionaries and cannibals" problem [Amarel 1968]. The general idea of chunking can ultimately be traced back to Miller's psychological studies [Miller 1956].

1.3 Relation to Other Machine Learning Research

EBL is characterized by the fact that it makes use of extensive background knowledge to guide the learning process. A number of researchers outside the area of EBL have also used such knowledge-intensive approaches to machine learning. Some early examples include Lenat's AM program [Lenat 1982], Sussman's HACKER program [Sussman 1975], and Soloway's program for learning rules of competitive games [Soloway 1978]. These systems are difficult to compare since they use diverse program architectures. Their background knowledge is embedded in specialized, domain-dependent heuristics, such as Lenat's heuristics for creating and evaluating concepts and Sussman's knowledge base of bugs and patches. Additional programs using knowledge-intensive learning techniques include Buchanan and Mitchell [1978], Vere [1977], Lebowitz [1983], Stepp and Michalski [1986], and Lenat et al. [1986].

The search control technique known as "dependency-directed backtracking" (DDB) provides an interesting comparison to EBL. This technique is used to control the process of backtracking when a contradiction or failure is encountered during a search process [Doyle 1979; Stallman and Sussman 1977]. DDB may also be interpreted as a type of explanation-based learning. DDB uses data dependencies to generalize the context of a contradiction, or search failure, in much the same manner that EBL uses explanations to generalize from training examples.

Attempts at formally classifying the types of background knowledge useful for inductive learning have been undertaken by both Michalski and Russell. Michalski [1983] developed a typology describing various kinds of "problem background knowledge" that can be used by inductive learning systems. Russell [1986] has attempted to exhaustively identify the types of information that can enable a system to make deductively sound generalizations. Natarajan and Tadepalli [1988] have developed a framework for analyzing the impact of background knowledge on the information complexity of learning from examples.

2. SELECTED EXAMPLES OF EXPLANATION-BASED LEARNING

2.1 Introduction

The techniques of explanation-based learning can be understood in a number of different ways. As described above, EBL represents a merging of several trends in machine learning research. These include research into generalization, chunking, operationalization, and analogy. Each of these research areas has contributed a distinct view of EBL. In this section EBL programs are classified in terms of these four categories. The category for each system is chosen to reflect the language used by its authors in describing their work. In many cases the differences between systems in separate categories are only a matter of interpretation. Programs described differently by their authors often involve similar underlying procedures. The authors have merely chosen to emphasize different aspects of their work or different ways of

Mapping justified generalization to chunking:

Explanation rules	→ Operators,
Explanation	→ Operator sequence,
Generalized explanation	→ Compiled operator sequence,
Example	→ Problem state
	→ Instantiated operator sequence,
Learned concept	→ Precondition of operator sequence
	→ Generalized operator sequence.

Mapping justified generalization to operationalization:

Explanation rules	→ Nonoperational concept description,
Explanation	→ Translation process,
Generalized explanation	→ Compiled translation process,
Example	→ Example,
Learned concept	→ Operational concept description.

Mapping justified generalization to justified analogy:

Explanation rules	→ Causality rules
	→ Problem-solving derivation rules,
Explanation	→ Network of causal relations
	→ Derivation of solution,
Generalized explanation	→ Transferred causal subnetwork
	→ Transferred portion of derivation,
Example	→ Analog or target,
Learned concept	→ Concept including analog and target.

Figure 3. Relation among views of EBL.

thinking about their programs. In this section an attempt is made to show how most EBL programs can be understood from each of the four points of view. Figure 3 suggests some rough correspondences between the different views of EBL. The reader should refer back to this figure while reading about each program.

2.2 EBL = Justified Generalization

Explanation-based learning is most often viewed as a method of generalizing from examples. As described above, the generalization process is usually framed in terms of a two-step procedure: (1) Explain the example, and (2) analyze the explanation in order to generalize the example. Figure 3 shows five roles that are a part of this process, including "explanation rules," "explanations," "generalized explanations," "examples," and "learned concepts." When reading about EBL generalization programs, it is useful to keep the two-step process in mind, and to consider how each of the roles is filled in a particular program.

2.2.1 GENESIS (DeJong and Mooney)

One of the major efforts to investigate EBL has been undertaken by DeJong and co-workers at the University of Illinois [DeJong 1986; DeJong and Mooney 1986; Mooney and DeJong 1985; O'Rorke 1984; Segre and DeJong 1985; Shavlik 1985]. The GENESIS system is a typical example of their work [Mooney 1985; Mooney and DeJong 1985]. GENESIS has been presented by DeJong and Mooney as a system for generalizing examples. It is intended to investigate explanation-based learning in the domain of human problem-solving behavior. GENESIS reads natural language stories that describe people engaged in carrying out plans to achieve typical human goals. It attempts to generalize from the stories to form schemata describing general plans for achieving goals. A story of a kidnapping is shown in Figure 4. GENESIS is able to generalize this single example of a kidnapping into a schema describing a generalized plan for kidnapping to obtain ransom. The schema contains only those

Fred is the father of Mary and is a millionaire. John approached Mary. She was wearing blue jeans. John pointed a gun at her and told her he wanted her to get into his car. He drove her to his hotel and locked her in his room. John called Fred and told him John was holding Mary captive. John told Fred if Fred gave him $250,000 at Trenos then John would release Mary. Fred gave him the money and John released Mary.

Figure 4. A story that GENESIS reads and generalizes [Mooney 1985].

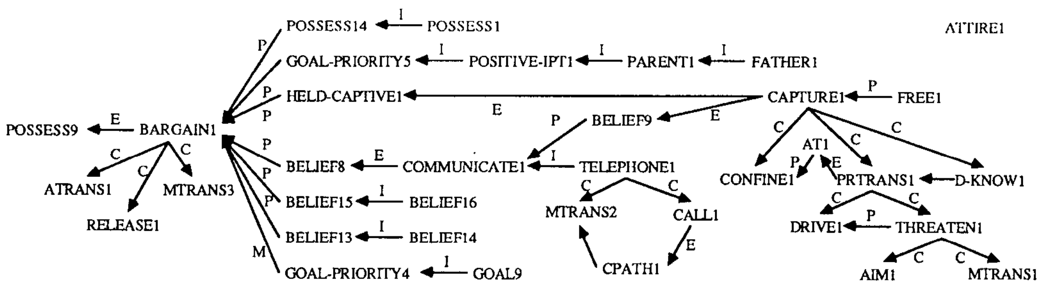
elements of the story that were necessary for the kidnapping to be successful, but none of the extra details. For instance, the schema requires that the victim be someone who is in a close personal relationship with a rich person since this constraint is necessary for the kidnapping to succeed. It does not require that the victim be wearing blue jeans or that the money be delivered at Trenos, since the success of the plan does not depend on these details.

In order to generalize a story, GENESIS builds a "causally complete explanation" of the events the story describes. Although the story describes a sequence of events, it does not state the causal connections between events. GENESIS must infer these connections. A causally complete description of the kidnapping story is shown in Figure 5. In the course of building this explanation, the system had to make several types of inferences. All of the "support links" (effects, preconditions, motivations, and inferences) [Mooney 1985] and "component links" were inferred by the system. For example, GENESIS inferred that the telephone call fulfilled a precondition for the bargain made between John and Fred. The system also inferred that the actors in the story had certain goals or goal priorities, for example, that Fred wanted Mary to be safe more than he wanted to keep his $250,000. In addition, the system inferred that certain actions in the story were components of composite plans, for example, that the action of pointing a gun is part of a "threaten" plan, which itself is part of a "capture" plan. The explanation is "complete" in the sense that all volitional actions are understood to be motivated by typical human goals, that is, "thematic goals" [Mooney 1985; Schank and Abelson 1977]. Each action achieves a thematic goal directly or else is part of a plan that fulfills a thematic goal.

In order to build explanations of stories, GENESIS draws upon a knowledge base containing facts about typical human goals and motivations. The knowledge base also describes actions and plans for achieving such goals. This knowledge is organized into a hierarchy of schemata describing actions, states, and objects. The actions are represented in a manner similar to STRIPS-type operators [Fikes et al. 1972]. Each action has a list of preconditions and a list of effects. GENESIS uses a combination of script-based [Cullingford 1978] and plan-based [Wilensky 1978] story-understanding methods [Schank and Abelson 1977]. Script-based methods operate by instantiating general schemata to match observed action sequences. Plan-based methods require searching a space of goals and plans to find those that would account for peoples' actions.

The GENESIS generalization process is charged with the task of building a schema describing plans for a wide variety of situations. For this purpose the generalizer analyzes the explanation of the story to determine which aspects are essential to the plan and which are irrelevant. The generalizer removes as much information from the story as possible, as long as the explanation of the success of the plan remains valid. If the explanation remains valid, the generalized plan should also be successful. Therefore, this procedure may be said to produce justified generalizations. The generalization procedure is shown in Figure 6.[3]

[3] The GENESIS system has apparently gone through more than one implementation. Two similar generalization procedures are described by Mooney [1985] and DeJong and Mooney [1986]. The procedure described here is essentially the one in DeJong and Mooney [1986], except that one step has been omitted. The omitted step requires replacing observed inefficient subplans with more efficient subplans, when possible. Two additional steps are mentioned in Mooney [1985]. One step involves "constraining the achieved goal to be thematic," and the other step involves enforcing a constraint that all generalized schemata be "well formed."



POSSESS9	John has $250,000.
BARGAIN1	John makes a bargain with Fred in which John releases Mary and Fred gives $250,000 to John.
MTRANS3	John tells Fred he will release Mary if Fred gives him $250,000.
RELEASE1	John releases Mary.
ATRANS1	Fred gives John $250,000.
POSSESS14	Fred has $250,000.
POSSESS1	Fred has millions of dollars.
GOAL-PRIORITY5	Fred wants Mary free more than he wants $250,000.
POSITIVE-IPT1	Fred has a positive interpersonal relationship with Mary.
PARENT1	Fred is Mary's parent.
FATHER1	Fred is Mary's father.
HELD-CAPTIVE1	John is holding Mary captive.
CAPTURE1	John captures Mary.
D-KNOW1	John finds out where Mary is.
PTRANS1	John moves Mary to his hotel room.
DRIVE1	John drives Mary to his hotel room.
THREATEN1	John threatens to shoot Mary unless she gets in his car.
AIM1	John aims a gun at Mary.
MTRANS1	John tells Mary he wants her to get in his car.
AT1	Mary is in John's hotel room.
CONFINE1	John locks Mary in his hotel room.
FREE1	Mary is free.
BELIEF8	Fred believes John is holding Mary captive.
COMMUNICATE1	John contacts Fred and tells him he is holding Mary captive.
TELEPHONE1	John calls Fred and tells him he is holding Mary captive.
CALL1	John calls Fred on the telephone.
CPATH1	John has a path of communication to Fred.
MTRANS2	John tells Fred he has Mary.
BELIEF9	John believes he is holding Mary Captive.
BELIEF15	John believes Fred has $250,000.
BELIEF16	John believes Fred has millions of dollars.
BELIEF13	John believes Fred wants Mary free more than he wants $250,000.
BELIEF14	John believes Fred is Mary's father.
GOAL-PRIORITY4	John wants to have $250,000 more than he wants to hold Mary captive.
GOAL9	John wants to have $250,000.
ATTIRE1	Mary is wearing blue jeans.

Link Types	Definition
P = Precondition	A state may be a precondition for an action.
E = Effect	A state may be an effect of an action.
I = Inference	The occurrence of one state or action implies the occurrence of another.
C = Component	An action may be a component of a plan.
M = Motivation	A goal state may motivate an action.

Figure 5. A causally complete explanation of the kidnapping [DeJong and Mooney 1986].

1. Delete parts of the story representation that are not essential to the explanation.
   (a) Remove parts of the network that do not causally support the achievement of the main thematic goal.
   (b) Remove nominal instantiations of known schemata.
   (c) Remove actions and states that only support inferences to more abstract actions or states.
2. Generalize the remaining schemata while maintaining the validity of each support link.
   (a) Extract the explanation structure ES from the explanation network.
   (b) Find the most general instantiation of ES that represents a valid explanation (EGGS procedure).
3. Package the generalized network into a schema.

Figure 6. Generalization procedure used by GENESIS.

The first part of GENESIS' generalization procedure is directed toward isolating the essential parts of the explanation (Step 1 in Figure 6). Some portions of the network representation of the story are not considered to be parts of the explanation per se and are pruned away by the system. To begin with, the system removes all actions and states that are not topologically connected through "support" or "component" links to the main thematic goal (Step 1a in Figure 6). These nodes are removed because they do not causally contribute to the achievement of the goal. In the network of Figure 5, the node asserting that Mary was wearing blue jeans is removed for this reason. The system also prunes the nodes describing actions that are mere "nominal instantiations" of known composite schemata (Step 1b in Figure 6). These constituent actions do not contribute to the main thematic goal except through the effects of the corresponding composite schemata. Since the composite schemata remain unpruned, the constituents are not needed. In the network of Figure 5, component actions of the "telephone" and "capture" schemata are removed.

The final pruning step depends crucially on the fact that all action and state schemata are organized into an "isa" hierarchy. All inferences of the form "Schema A is an instance of Schema B" are deleted from the explanation, whenever "Schema A" serves no purpose other than supporting the inference to "Schema B" (Step 1c in Figure 6). For example, in Figure 5 the inferences that the "father" relationship is an instance of "parent," which is itself an instance of "positive-ipt," are deleted along with the "father" and "parent" nodes. These nodes are deleted since they are not needed to support the "goal-priority" node. The goal-priority node was created using an inference rule inherited from the "positive-ipt" node. Since this rule applies to relationships more general than "father" or "parent," these two nodes are overspecific and must be deleted. This step of the generalization process also leads to deleting the "telephone" node and the inference that the telephone action is an instance of the "communicate" schema.

After the nonessential parts of the explanation are pruned away, the next step is to generalize the remaining schemata (Step 2 in Figure 6). The slot fillers on the remaining schemata are generalized as much as possible as long as the support links remain valid. Each support link was created by using some general inference rule from the knowledge base. While building the explanation, GENESIS annotated the support links with pointers to the inference rules from which the links were created. In order for the support links to remain valid, the schemata can only be generalized in such a way that they continue to match the patterns in the general inference rules.

The schemata are generalized in a two-step process (Steps 2a and 2b in Figure 6). GENESIS first extracts the so-called **explanation structure** from the explanation network. The explanation structure may be defined as the result of replacing each support link in the network with the associated general inference rule [Mitchell et al. 1986; Mooney and Bennett 1986]. The explanation structure represents an overgeneralized version of the original explanation. In the second step, GENESIS uses a procedure called **EGGS** to specialize the explanation structure [DeJong and Mooney 1986; Mooney and Bennett 1986]. An outline of the EGGS algorithm is shown in Figure 7.[4]

The EGGS procedure takes an explanation structure ES as its input. EGGS is charged with the task of finding the most general instantiation of ES that represents a valid explanation. In order that ES represent a valid explanation, the rule patterns must be reinstantiated to some degree. In particular, if R1 and R2 are two rules incident on a given node of ES, then the appropriate patterns from R1 and R2 must be instantiated to syntactically identical expressions.[5] EGGS first forms a list of all pairs of patterns that must be instantiated to identical expressions. Then EGGS finds the maximally general set of bindings for the pattern variables that will simultaneously unify all equated pairs of patterns. Finally, all the rule patterns in ES are instantiated with these bindings. The resulting network is the most general instantiation of ES that represents a valid explanation.

After the EGGS procedure is applied to the kidnapping explanation, some of the objects are generalized and others are constrained. For example, the locations in the "hold-captive" and "bargain" schemata are generalized. The amount of money is constrained to be any amount possessed by the target of the kidnapping. The victim of the "capture" schema is constrained to be the same person mentioned in the "positive-ipt" schema. The resulting generalized explanation network is shown in Figure 8.

The final step of GENESIS' generalization procedure requires packaging the generalized network into an action schema (Step 3 in Figure 6). The resulting schema contains "preconditions," "effects," and

```
Given: An explanation structure ES.

Find:  The most general instantiation of ES that
       represents a valid explanation.

Procedure:
1. Let L be a list of all pairs of inference rule patterns
   in ES that must be instantiated to syntactically
   identical expressions.
2. Initialize S to be the null substitution.
3. For each pair (A, B) of equated patterns on the list
   L do:
   (a) Let A′ be the result of applying S to A.
   (b) Let B′ be the result of applying S to B.
   (c) Let T be the most general unifier of A′ and B′.
   (d) Let S be the composition of substitutions S and
       T.
4. For each inference rule pattern P in ES do:
   (a) Let P′ be the result of applying S to P.
   (b) Modify ES by replacing P with P′.
```

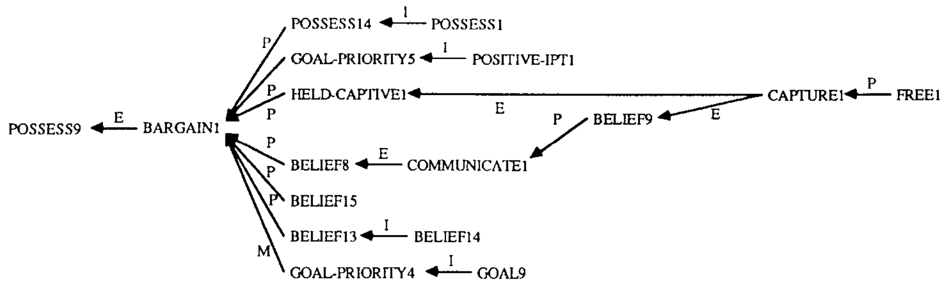
Figure 7. EGGS procedure.

"expansion schemata." The generalized preconditions include just the precondition nodes in the network that are not created as effects of any other node in the network. The generalized effects include the effect nodes in the network that are not undone by any other action in the network. The generalized expansion schemata are all the remaining nodes in the generalized explanation.

The results of the GENESIS learning process can be evaluated in terms of the system's question-answering ability [Mooney 1985]. Before building the schema, GENESIS reads a test narrative and is unable to answer some questions about the narrative. In order to answer the questions, GENESIS would have to make some default inferences. The inferences could be made by a plan-based story understander; however, GENESIS has only a rudimentary capability for plan-based story understanding and is unable to make the necessary inferences. After forming a generalized kidnapping schema, GENESIS can read the narrative and successfully answer the very same questions. GENESIS is able to make the necessary inferences by using the generalized schema in a script-based story understanding process. One can summarize the results of learning in GENESIS in the

[4] The algorithm shown in Figure 7 most closely resembles the version of EGGS presented by Mooney and Bennett [1986]; however, Mooney and Bennett's presentation contains a typographic error, omitting Steps 3a and 3b shown in Figure 7.

[5] An additional constraint is mentioned by Mooney [1985]. This constraint requires that the pattern representing the main goal of the story match a thematic goal pattern.



POSSESS9	Person1 has Money1.
BARGAIN1	Person1 makes a bargain with Person2 in which Person1 releases Person3 and Person2 gives Money1 to Person1.
POSSESS14	Person2 has Money1.
GOAL-PRIORITY5	Person2 wants Person3 free more than he wants Money1.
POSITIVE-IPT1	There is a positive interpersonal relationship between Person2 and Person3.
HELD-CAPTIVE1	Person1 is holding Person3 captive.
CAPTURE1	Person1 captures Person3.
FREE1	Person3 is free.
BELIEF8	Person2 believes Person1 is holding Person3 captive.
COMMUNICATE1	Person1 contacts Person2 and tells him he is holding Person3 captive.
BELIEF9	Person1 believes he is holding Person3 captive.
BELIEF15	Person1 believes Person2 has Money1.
BELIEF13	Person1 believes Person2 wants Person3 free more than he wants Money1.
BELIEF14	Person1 believes there is a positive interpersonal relationship between Person2 and Person3.
GOAL-PRIORITY4	Person1 wants to have Money1 more than he wants to hold Person3 captive.
GOAL9	Person1 wants to have Money1.

Figure 8. The generalized kidnapping network [DeJong and Mooney 1986].

following way: Before learning, the knowledge base is suitable only for use by a plan-based understanding system. After learning, the knowledge base contains new schemata that can be used by a script-based system. This represents an improvement because plan-based understanding requires more search than is necessary for script-based understanding [Wilensky 1978].

Although GENESIS has been presented as a program that learns by generalizing examples, it can also be viewed in other ways (see Figure 3). It can be regarded as a chunking system, which learns by combining operators into macro operators. The generalized kidnapping schema may be viewed as a macro operator composed of the "capture," "communicate," and "bargain" operators. GENESIS may also be viewed as a system that reformulates nonoperational concept descriptions. Before learning, the system may be said to possess a nonoperational description of the concept "plans for obtaining money." The pattern describing the thematic goal "obtain money," together with the knowledge base of action schemata, could be viewed as a nonoperational specification of the collection of all plans for obtaining money. The description is nonoperational since the information about what constitutes a valid plan for obtaining money is scattered throughout the knowledge base. After learning, GENESIS has an operational description of the concept in the form of a general schema. The schema explicitly describes a set of plans. Any instantiation of the generalized schema is a valid plan for obtaining money.

2.2.2 *LEX-II (Mitchell and Utgoff)*

Another major effort to investigate EBL techniques was undertaken by Mitchell and co-workers at Rutgers University. A number of different EBL systems were developed by Mitchell's group [Kedar-Cabelli 1985; Keller 1983; Mahadevan 1985; Mitchell 1982b; Mitchell 1983; Mitchell et al. 1986; Steinberg and Mitchell 1985; Utgoff 1986]. One of the oldest of these systems is LEX-II, which learns search control heuristics in the domain of symbolic integration. LEX-II was built as an extension to the LEX-I system. LEX-I uses purely empirical techniques for learning concepts from multiple examples [Mitchell et al. 1981; Mitchell et al. 1983a]. LEX-II was built to combine the empirical techniques of LEX-I with analytical (EBL) learning methods for generalizing from single examples [Mitchell 1982b, 1983].

LEX-I and LEX-II both contain four main modules: the problem generator, the problem solver, the critic, and the generalizer. The problem solver is equipped with a set of operators that it uses in a best-first forward search process. Sample operators are shown in Figure 9. Each operator has a condition specifying the class of problem states to which the operator can be "validly" applied. The learning modules are charged with the task of finding more restrictive conditions on the states to which the operators will be applied. For each operator the system tries to learn a concept describing the set of states to which the operator can be "usefully" applied in order to find a solution. The states to which an operator can be "usefully" applied is usually a proper subset of those states to which it can be "validly" applied. The restricted applicability conditions limit the number of states created, leading to a faster search process and increasing the range of problems that the system can solve within a fixed time limit.

The learning process begins when a problem is created by the problem generator. The problem solver attempts to solve the problem. If a solution is found, a trace of the search tree is sent to the critic module. The critic labels some or all of the operator applications in the search tree as being "useful" or "not useful." The operator applications along the final solution path are considered "useful," and those that lead away from the final solution path are considered to be "not useful" (Figure 10). The classification yields sets of positive and negative instances for each operator. These examples are used by the generalizer in order to learn restricted operator applicability conditions.

LEX-I processes the labeled instances using Mitchell's candidate elimination algorithm [Mitchell 1978], a purely empirical concept-learning technique. This algorithm searches through a "version space" containing an initial set of candidate concept descriptions. The candidates are connected

OP1: $$\int \sin(x)\,dx \rightarrow c - \cos(x).$$

OP2: $$f(x)^r \rightarrow f(x)\,f(x)^{(r-1)}.$$

OP3: $$\int r\,f(x)\,dx \rightarrow r \int f(x)\,dx.$$

OP4: $$\int x^{(r\neq-1)}\,dx \rightarrow \frac{x^{(r+1)}}{(r+1)} + c.$$

Figure 9. Examples of operators used in LEX-I and LEX-II.

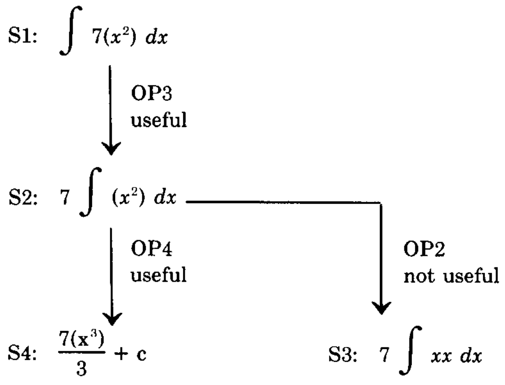

$$\begin{tikzcd}
S1{:}\ \int 7(x^2)\,dx \arrow[d, "\text{OP3 useful}"] & \\
S2{:}\ 7\int (x^2)\,dx \arrow[d, "\text{OP4 useful}"] \arrow[dr, "\text{OP2 not useful}"] & \\
S4{:}\ \frac{7(x^3)}{3} + c & S3{:}\ 7\int xx\,dx
\end{tikzcd}$$

Figure 10. Partial search tree labeled by critic module.

$(\forall op, s)\{POSINST(op, s) \Leftarrow \text{USEFUL}(op, s)\}.$
$(\forall op, s)\{\text{USEFUL}(op, s) \Leftarrow [\neg\text{SOLVED}(s) \land \text{SOLVABLE}(\text{APPLY}(op, s))]\}.$
$(\forall op, s)\{\text{SOLVABLE}(s) \Leftarrow \text{SOLVABLE}(\text{APPLY}(op, s))\}.$
$(\forall op, s)\{\text{SOLVABLE}(s) \Leftarrow \text{SOLVED}(\text{APPLY}(op, s))\}.$

Figure 11. Rules defining the POSINST predicate in LEX-II.

by "generalization-of" and "specialization-of" relations, which define a lattice.[6] As each positive or negative instance is processed, the algorithm eliminates all candidates that are inconsistent with the critic's classification of the example. This is achieved by recording two sets called "S" and "G." These sets respectively contain the maximally specific (S) and maximally general (G) candidates that are consistent with all the instances observed so far. When observing a positive instance, each member of S is generalized just enough to include the new example. Negative instances are processed by specializing each member of G just enough to exclude the new example. If a sufficient number of examples is observed, the sets S and G converge to contain only one possible candidate, assuming that the correct concept description is actually contained in the version space.

LEX-II was developed with the intention of providing the learning modules with additional forms of knowledge that would enhance the effectiveness of the generalization process. In particular, LEX-II was given a description of the goal of the learning process. LEX-II contains rules that provide an abstract definition of a positive instance predicate, POSINST, shown in Figure 11. These rules provide definitions of a whole collection of concepts, one concept for each operator. For example, when the rules are instantiated by letting "*op*" equal "OP3," they define the concept including the set of states that satisfy "POSINST(OP3, *s*)." To paraphrase these rules, a state "*s*" is a positive instance for the operator "*op*" if the state "*s*" is not already solved and applying "*op*" to "*s*" leads to a state that is either solved or is solvable by additional operator applications.

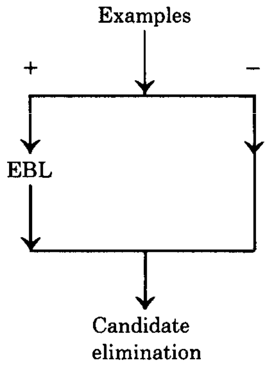


Figure 12. Organization of the LEX-II generalizer.

The organization of the LEX-II learning module is shown in Figure 12. LEX-II processes positive and negative examples somewhat differently. The positive examples are submitted to an EBL procedure. The EBL procedure generalizes single positive instances by making use of the rules defining the POSINST predicate. The generalized positive instances are then submitted to the candidate elimination procedure. The negative instances are submitted directly to the candidate elimination algorithm, just as they are in LEX-I.

The EBL procedure used in LEX-II is shown in Figure 13. The input to this procedure is a state and operator application pair, (OP, S), that was labeled as a positive instance by the critic module. The EBL procedure is charged with finding a generalization of S representing a set of states for which the operator OP will be "useful." The LEX-II EBL procedure uses a two-step process similar to the one described above for the GENESIS system. First

[6] The version space is defined by a context-free grammar. Each sentential form in the language of the grammar corresponds to a learnable concept. The rules of the grammar correspond to the relations "generalization of" and "specialization of," which define the lattice.

Given: An example state and operator application pair, (OP, S), that was classified as a positive instance.

Find: A generalization of S representing a set of states for which the operator OP will be "useful."

Procedure:

1. Build an explanation showing how the pair, (OP, S), satisfies the definition of the POSINST predicate.
2. Analyze the explanation in order to obtain a set of conditions sufficient for any state "*s*" to satisfy the predicate "POSINST(OP, *s*)."
   (a) Change the state constant "S" into a universally quantified variable "*s*."
   (b) Extract a set of conditions satisfying the AND/OR tree representing the explanation.
   (c) Translate the conditions from the "operator language" into the "generalization language."
      (1) Express the conditions in terms of restrictions on various states in the solution tree.
      (2) Propagate restrictions through the solution tree to obtain equivalent restrictions on the example problem state "*s*."

Figure 13. EBL procedure used in LEX-II.

$$\begin{tikzcd}
 & \text{POSINST(OP3, S1)} & \\
 & \text{USEFUL(OP3, S1)} \arrow[u] & \\
\text{SOLVABLE(APPLY(OP3, S1))} \arrow[ur] & & \neg\text{SOLVED(S1)} \arrow[ul] \\
\text{SOLVED(APPLY(OP4, APPLY(OP3, S1)))} \arrow[u] & &
\end{tikzcd}$$

Figure 14. Proof tree built by LEX-II.

LEX-II explains why the example is a positive instance. Then LEX-II generalizes the example by analyzing the explanation.

In order to illustrate the LEX-II EBL procedure, consider an example from the labeled search tree shown in Figure 10. The labeling indicates that the pair (OP3, S1) is a positive instance. LEX-II begins processing this instance by verifying that the pair (OP3, S1) does indeed meet the conditions given in the definition of the POSINST predicate. LEX-II verifies this example by building the AND/OR proof tree shown in Figure 14. The root of the explanation tree asserts that the example pair (OP3, S1) is a positive instance. The leaves of the tree represent the facts on which the explanation is based. These leaf nodes make assertions about the structure of the search tree shown in Figure 10.

After building an explanation verifying that (OP3, S1) is a positive instance, LEX-II analyzes the explanation in order to generalize the state S1. The first two steps involve (1) changing the state constant "S1" in the example into a universally quantified variable "*s*"[7] and (2) extracting a set of nodes sufficient to satisfy the AND/OR proof tree (Steps 2a and 2b in Figure 13). Any set of nodes satisfying the AND/OR proof tree would constitute sufficient conditions. In practice, however, only leaf nodes are chosen. For the explanation tree in Figure 14, LEX-II forms the following clause:

$$(\forall s)\{\text{POSINST}(\text{OP3}, s) \Leftarrow [\text{SOLVED}(\text{APPLY}(\text{OP4}, \text{APPLY}(\text{OP3}, s))) \wedge \neg\text{SOLVED}(s)]\}.$$

[7] For this step to be "justified" LEX-II should consult the definitions of the rules used in the proof to verify that the proof remains valid after the state constant is changed to a variable; however, Mitchell does not mention such a process.

This clause provides a set of conditions specifying when the operator OP3 can be usefully applied to a state. The conditions are sufficient, but not necessary, for a state to be a member of the set "states for which OP3 is useful." The conditions are sufficient since the explanation tree will be satisfied by any state meeting these conditions. The conditions are not necessary, however, because there are other explanation trees that could prove the same result.

The antecedents of the clause are written in the so-called "operator language" of LEX-II. In this form they are not particularly useful because they can only be tested by applying operators to states and examining the results. LEX-II makes the clause more useful by translating the antecedent conditions into the "generalization language"[8] (Step 2c in Figure 13). LEX-II begins the translation by applying the definitions of "SOLVED" and "¬SOLVED." By substituting definitions of these predicates, LEX-II obtains a conjunction of statements of the form MATCH(⟨generalization⟩, ⟨state⟩), where "⟨generalization⟩" is a statement in the generalization language and "⟨state⟩" can refer to any state in the symbolic integration state space. When these substitutions are applied to the clause shown above, the following result is obtained:

$$\begin{aligned}&(\forall s)\{\mathrm{POSINST}(\mathrm{OP3}, s)\\ &\Leftarrow [\mathrm{MATCH}(\langle \text{function}\rangle, \mathrm{APPLY}(\mathrm{OP4}, \mathrm{APPLY}(\mathrm{OP3}, s)))\\ &\quad \wedge \mathrm{MATCH}(\textstyle\int \langle \text{function}\rangle\, dx, s)]\}.\end{aligned}$$

This clause contains references to several states in the search tree. The states are described by sequences of operator applications as indicated by the APPLY function. The final translation step removes the references to operator applications to obtain conditions expressed directly in terms of the example state "*s*." In order to remove references to the APPLY function, a procedure called **constraint back-propagation** (CBP) is used [Utgoff 1986]. The CBP technique is given the task of translating any statement in the form MATCH(P, APPLY(OP, *s*)) into an equivalent statement of the form MATCH(P′, *s*). This is essentially equivalent to calculating "weakest preconditions" as formalized in Dijkstra [1976] and to performing **goal regression** [Nilsson 1980; Waldinger 1977]. The pattern P′ must meet the requirement that a state S will match P′ if and only if the state APPLY(OP, S) matches P. The CBP procedure is implemented by writing one LISP function for each problem-solving operator. The LISP function represents the "inverse" of that operator.[9] The inverse for operator OP would take a pattern such as P and find the corresponding weakest precondition P′.[10] After applying the CBP procedure to the antecedents in the clause shown above, the following result is obtained:

$$\begin{aligned}&(\forall s)\{\mathrm{POSINST}(\mathrm{OP3}, s)\\ &\Leftarrow [\mathrm{MATCH}(\textstyle\int r(x^{\{r\neq -1\}})\, dx, s)\\ &\quad \wedge \mathrm{MATCH}(\textstyle\int \langle \text{function}\rangle\, dx, s)]\}\end{aligned}$$

The power of the LEX-II generalization procedure can be illustrated by comparing this result to the original example shown in Figure 10. The original example only asserted the usefulness of applying operator OP3 to the single problem state S1. The clause shown above asserts the usefulness of applying operator OP3 to a larger class of problem states. Two distinct generalizations have been made. The coefficient "7" has been generalized to "*r*," any real number. Furthermore, the exponent "2" has been generalized to any real number "r," other than "−1."

[8] The generalization language is specified by the same context-free grammar that defines the version space.

[9] Strictly speaking, these LISP functions are not true inverses of the corresponding operators. If OP maps problem states to problem states, the true inverse would map states to states. The so-called "inverse" used here maps patterns (sets of states) to other patterns.

[10] A difficulty arises when the precondition P′ cannot be expressed in the generalization language of LEX-II. When this happens, the system defines new terms to expand the generalization language so that it can express the desired precondition [Utgoff 1986]. On one occasion the system was led to define a new term equivalent to "odd integer" in order to resolve such an impasse.

The final generalization step involves combining the results of (EBL) generalization of single positive examples with the candidate elimination algorithm. The clause shown above provides sufficient (but not necessary) conditions for concept membership. Therefore, every candidate concept description must be at least as general as this generalized positive instance. For this reason, the generalized instance can be processed by the candidate elimination algorithm just as if it were an actual positive instance. The algorithm must simply generalize each member of the boundary set S just enough to include the generalized positive instance.

Although LEX-II does not use its EBL techniques to process negative examples, there is no reason in principle why this cannot be done. The system could be provided with a set of rules for proving statements of the form ¬POSINST(OP, S). By processing explanations of negative instances, the system could obtain generalized negative instances. These could be used to refine the boundary set G, just as generalized positive instances are used to refine the boundary set S. In practice, explanations of negative instances might be large and difficult to analyze if the predicate POSINST is defined as above. A proof of ¬POSINST(OP, S) would require showing that the state APPLY(OP, S) is a dead end. This would mean proving that no operators apply or that all applicable operators lead to other dead-end states. In the worst cases, such proofs can require reasoning about a large number of states along multiple paths in the search tree. Proofs of POSINST(OP, S) need only reason about states along a single solution path.

The value of using EBL techniques in LEX-II can be assessed by observing the rate at which learning occurs. The candidate elimination algorithm should converge faster in LEX-II than in LEX-I, because LEX-II uses generalized positive instances to refine the boundary set S. The EBL techniques effectively provide a stronger bias for inductive learning. LEX-I makes use of the bias contained in the definition of the generalization language. LEX-II uses this bias in addition to the constraints provided by using EBL techniques to generalize positive instances. The stronger bias and faster rate of convergence should lead to improved performance by the problem solver, since the learned heuristics are available earlier in LEX-II than in LEX-I.

The learning component of LEX-II was able to improve the overall problem-solving performance of the system during the initial stages of learning. Eventually a point was reached after which the acquisition of new heuristics failed to improve the overall performance of the system [Mitchell 1983]. The difficulty results from the fact that the heuristics learned by LEX-II are capable of improving only some aspects of the system's performance. Heuristics help decide what operator to apply, given a state to be expanded. They do not provide direct guidance about what state should be chosen for expansion. Eventually the system's performance was limited by the decision of which state to expand, rather than which operator to apply. This "wandering bottleneck" problem results from the fact that only some aspects of the system's performance fall within the scope of the learning module.

Although LEX-II has been presented mainly as a system for generalizing from examples, it can also be viewed in other ways (see Figure 3). LEX-II can be viewed as a system that performs "chunking" of operator sequences to form macro operators. In the example shown above, the system learns a condition describing the set of states for which the sequence OP3 followed by OP4 will lead to a solved state. The system could save this macro along with its applicability condition. Although LEX-II does not actually save such macro operators, it could easily be extended to do so.

LEX-II can also be viewed as a system that reformulates nonoperational concept descriptions. In the example shown above, the system translates the concept "POSINST(OP3, *s*)" into a conjunction of patterns in the system's generalization language. LEX-II could, in principle, be modified to reformulate concepts other than the POSINST predicate defined in Figure 11. As described by Mitchell [1982b], the rules defining POSINST could be changed so that an operator application is considered to be useful only if it lies along a *minimum* cost path to a solution; however, this change was apparently never implemented. Were the rules so modified, they would probably lead to large and complex explanations that would be difficult to analyze, just as explanations for negative instances would be difficult to analyze. Proving a path to be minimal in cost would require reasoning about an entire search tree, rather than merely reasoning about a single solution path.

2.2.3 Similar Work

Several other investigators have developed EBL systems that are naturally viewed in terms of generalizing from examples. Minton [1984] implemented a variant of EBL called "constraint-based generalization." He used the method in a program that learns forced win positions in games like tick-tac-toe, go-moku, and chess. Two similar EBL systems that operate in the domain of logic circuit design were developed independently by Ellman [1985] and Mahadevan [1985]. Ellman's program is capable of generalizing an example of a shift register into a schema describing devices for implementing arbitrary bit permutations. The schema is created by a process that analyzes the proof of correctness of the example circuit. Mahadevan's method is called "verification-based learning" (VBL). The VBL technique is intended to be a general method of learning problem decomposition rules. Mahadevan has tested VBL in the domains of logic circuit design and symbolic integration. Hill [1987] has developed similar methods for the domain of software design. His system uses explanation-based methods to generalize program abstractions to promote software reuse.

A number of people working with DeJong developed EBL systems following up on GENESIS. O'Rorke built the "Mathematician's Apprentice" program [O'Rorke 1984, 1986], which uses explanation-based methods to create schemata summarizing successful theorem-proving episodes. Shavlik [1985, 1986] built a system that learns concepts from classical physics. His "PHYSICS 101" system learns concepts like conservation of momentum, starting with only a knowledge of Newton's laws and calculus. Segre developed a system that uses EBL methods to learn schemata describing robot manipulator sequences [Segre 1986; Segre and DeJong 1985].

2.3 EBL = Chunking

Chunking is usually understood in the context of problem spaces, problem states, and operators. A chunking system takes a linear or tree-structured sequence of operators as its input. The task of the chunking system is to convert the sequence of operators into a single "macrooperator," or "chunk," that has the same effect as the entire sequence. This process is sometimes described as "compiling" the operator sequence.

As shown in Figure 3, chunking can be placed into rough correspondence with the EBL generalization techniques described in the preceding section. The process of forming an operator sequence out of primitive operators is analogous to forming an explanation out of explanation rules. Compiling an operator sequence into a macro corresponds to analyzing and generalizing an explanation. Problem states may be seen to play the role of training examples. The chunking process produces a precondition for the macro operator. The macro precondition represents a generalization of the example state. It is also possible to view an instantiated operator sequence as a training example and view a generalized operator sequence as the learned concept.

2.3.1 SOAR (Laird, Newell, and Rosenbloom)

The SOAR project is an ambitious attempt to build a system combining learning and problem-solving capabilities into an architecture for general intelligence [Laird et al. 1986a, 1987]. The problem-solving methods in SOAR are based on "universal subgoaling" (USG) [Laird 1984] and the "universal weak method" (UWM) [Laird and Newell 1983a, 1983b]. Universal subgoaling is a technique for making all

$$\begin{tikzcd}
\text{(GOAL, SPACE, STATE, OPERATOR)} \\
\text{(SUBGOAL, SPACE, STATE, OPERATOR)} \arrow[u] \\
\text{(SUBSUBGOAL, SPACE, STATE, OPERATOR)} \arrow[u]
\end{tikzcd}$$

Figure 15. Hierarchy of goal contexts in SOAR.

search control decisions in a uniform manner. The universal weak method is an architecture that provides the functionality of all the **weak methods** [Newell 1969]. The learning strategy of SOAR is based on the technique for "chunking" sequences of production rules that was developed by Rosenbloom and Newell [1986; Rosenbloom 1983]. The developers of SOAR have put forward the hypothesis that chunking is a universal learning method. They also believe that chunking techniques are especially powerful when combined with the USG and UWM architecture.

The architecture of SOAR is based on the "problem space hypothesis" [Newell 1980], the notion that all intelligent activity occurs in a problem space. This idea is embodied in SOAR by allowing all decisions to be made in a single uniform manner, that is, by searching in a problem space. At any point in time, SOAR is working in a "current context" that describes the status of the search in whatever problem space SOAR is currently using. More specifically, the current context consists of four parts: a goal, a space, a state, and an operator. The current context can be linked to previous contexts so that a goal and subgoal hierarchy is formed (Figure 15). The components of each context are annotated with additional information called "augmentations." The hierarchy of contexts and associated augmentations make up the "working memory" of SOAR.

SOAR uses a special mechanism for controlling search in problem spaces. Production rules contained in "long-term memory" are charged with the task of deciding which one of the four items in the current context should be changed and how it should be changed. There are four types of possible changes, corresponding to the four parts of the context:

- Change the operator to be applied to the current state.
- Change the current state to be expanded.
- Change the problem space used to solve the current goal.
- Change the current goal to some other goal.

The production rules make search control decisions in a two-phase process [Laird and Newell 1983a]. In the first, "elaboration" phase, all rules are applied repeatedly in parallel to the working memory. The rules assert "preferences" regarding which part of the context should be changed and how it should be changed. In the second, "decision" phase, the preferences are tallied to see if a unique "best" choice is determined. When a unique best choice is determined, SOAR makes the change automatically.

Sometimes the production rules lack sufficient knowledge to make a search control decision. This problem is manifested when the decision phase fails to yield a unique best choice concerning how to change the current context. Under such circumstances the system is said to have reached an "impasse." For example, SOAR reaches a "tie impasse" when it cannot decide which of several operators should be applied to the current state. SOAR reaches a "no change impasse" when it does not know how to apply the current operator to the current state, because the operator is not directly implemented. An impasse is resolved in the same manner in which SOAR solves any other problem—by searching in a problem space. When the SOAR architecture detects that an impasse has occurred, it automatically sets up a subgoal and a new context to resolve the impasse. The "resolve-impasse" subgoal is solved in the usual way, by selecting a problem space, states, and operators. During processing of the subgoal, the system will hopefully accumulate sufficient information to make the search control decision that resulted in the impasse. In that case the subgoal gets terminated and SOAR returns to the original goal and context.

INITIAL STATE

2	3	1
	8	4
7	6	5

DESIRED STATE

1	2	3
8		4
7	6	5

Figure 16. Initial and goal states for 8-Puzzle [Laird et al. 1986a].

```
1. G1 solve-eight-puzzle
2. P1 eight-puzzle-sd
3. S1
```

2	3	1
	8	4
7	6	5

```
 4. O1 place-blank
 5. ==>G2 (resolve-no-change)
 6.       P2 eight-puzzle
 7.       S1
 8.       ==>G3 (resolve-tie-operator)
 9.            P3 tie
10.            S2 {left, up, down}
11.            O5 evaluate-object(O2(left))
12.            ==>G4 (resolve-no-change)
13.                 P2 eight-puzzle
14.                 S1
15.                 O2 left
16.                 S3
```

2	3	1
8		4
7	6	5

```
17.       O2 left
18.       S4
19. S4
20. O8 place-1
```

Figure 17. Trace of SOAR execution on 8-Puzzle [Laird et al. 1986a].

In order to illustrate the behavior of SOAR, consider the following example of solving the 8-PUZZLE. The 8-PUZZLE involves moving tiles around on a rectangular grid. The initial state and goal state for the puzzle are shown in Figure 16. In order to solve the puzzle, one must find a sequence of tile moves that transforms the initial state into the goal state. A partial trace of SOAR's solution is shown in Figure 17. SOAR starts with the goal "SOLVE-EIGHT-PUZZLE." An abstract problem space called "EIGHT-PUZZLE-SD" is selected. The operators of the abstract space are called "PLACE-BLANK," "PLACE-1," "PLACE-2," etc. Each such abstract operator is intended to achieve the function of moving one particular tile or the space to its goal position. SOAR chooses the operator "PLACE-BLANK" first. A "NO-CHANGE" impasse occurs because the abstract operator is not implemented and SOAR does not know how to apply it to the current state. A "RESOLVE-NO-CHANGE" goal is created to resolve the impasse. SOAR attempts to solve the new goal by working in the original "EIGHT-PUZZLE" problem space. Another impasse occurs later when SOAR cannot decide which of the three operators, "LEFT," "UP," or "DOWN," to apply. This leads to a new subgoal, and so on. The system eventually accumulates enough information to resolve the sequence of impasses and their associated subgoals. This occurs by line 16 when SOAR has tried applying the operator "LEFT" and discovers that the blank is now in its correct location. This means that SOAR has now found a way to apply the abstract operator "PLACE-BLANK" to this particular initial state.

The learning mechanism in SOAR is intended to acquire search control knowledge from problem-solving experience. In particular, the chunking system creates new production rules that help SOAR to make search control decisions more easily. The new rules enable SOAR to make such decisions directly through the elaboration and decision phases described above. The result is that fewer impasses occur and SOAR avoids the need to process subgoals. The chunking mechanism operates continuously. Whenever a subgoal terminates in SOAR, the chunking mechanism is invoked.[11] The mechanism attempts to build a new rule that will summarize the results of processing the subgoal. When the same subgoal occurs in an identical or similar situation, the new rule will fire and help make a decision that previously led to an impasse.

The chunking procedure is described by Laird et al. [1986a] and outlined in Figure 18. Assuming that a subgoal G has just successfully terminated, the chunking process will create a new production rule R having the same effect as the entire sequence of production rules that fired during the processing of goal G. The first step involves collecting conditions and actions for the new rule R. The conditions are found on a "referenced list" that was maintained during the processing of goal G. The "referenced list" contains all working memory elements that were created before goal G and were referenced by rules that fired during the processing of G. If these working memory elements were to be present in some other situation, they would enable the same sequence of rules to fire.[12] These working memory elements become the conditions of the new rule R. The actions of R are found by determining which working memory elements were created during processing of goal G and were passed on to supergoals of G by being attached as augmentations to the context of a supergoal of G. These actions are just the information that was remembered by the system after goal G was terminated. They constitute the information required to resolve the impasse that led to the creation of goal G.

In order that the new rule R apply to a variety of situations, some of the constants in the conditions and actions of R need to be generalized. In particular, the "identifiers" must be changed to variables. This is

Problem:

a. Given a goal G that has successfully terminated.
b. Create a new production rule R that has the same effect as the sequence of rules that were used to solve the goal G.

Procedure:

1. Collect conditions and actions.
   (a) Conditions of R include all working memory elements created before goal G that were referenced during processing of goal G.
   (b) Actions of R include all working memory elements created during processing of goal G that were passed on to supergoals of goal G.
2. Variabilization of identifiers.
   (a) All occurrences of a single identifier in R are changed to a single variable.
   (b) Occurrences of distinct identifiers in R are changed to distinct variables.
   (c) A condition asserting that distinct variables must match distinct identifiers is added to R.
3. Chunk optimization.

Figure 18. Chunking procedure in SOAR.

necessary since each identifier is unique to a working memory element. In order to choose variables, SOAR must determine which identifiers are required to be equal to each other and which are required to be distinct. The procedure shown in Figure 18 makes the decision in a conservative way, leading to chunk applicability conditions that are as restrictive as possible. It assumes that equal identifiers in the example are required to be equal and replaces them with a single variable. It also assumes that distinct identifiers in the example are required to be distinct and replaces them with distinct variables. An additional constraint is added to guarantee that distinct variables match distinct identifiers. Aside from some exceptional cases reported in [Laird et al. 1986b], the developers of SOAR claim this approach guarantees that no overgeneralized rules will be created, although overspecialized rules will sometimes be formed [Laird et al. 1986a]. The final step in Figure 18 involves making the new rule more efficient by reordering the conditions and making other changes for the sake of efficiency.

The chunking mechanism in SOAR has gone through several implementations. An

[11] This capability requires that the system meet the "goal-architecturality constraint"; that is, the representation of goals must be defined in the system architecture itself [Rosenbloom and Newell 1986].

[12] Strictly speaking, this requires that the system meet the "cryptoinformation constraint"; that is, the firing of rules must not be controlled by "hidden information" such as a conflict resolution strategy [Rosenbloom and Newell 1986].

earlier implementation used a different criterion for deciding when chunking should occur [Rosenbloom and Newell 1986]. The earlier criterion specified that chunking take place only for goals that were solved without invocation of subgoals. This resulted in "bottom-up" chunking, which was useful for cognitive modeling. A recent implementation uses a different method of finding the conditions that go into a rule created by the chunking mechanism [Laird et al. 1986a]. The new approach involves tracing dependencies from the results of a goal, through the sequence of rules that fired, back to working memory elements present before the goal was created. This approach leads to rules with greater generality than the one described above. The new method excludes conditions that were referenced by production rules that fired but did not contribute to the results of a goal because they led to dead ends.[13]

When the chunking mechanism is applied to the 8-PUZZLE problem, it generates a collection of rules that implement the abstract operators such as "PLACE-SPACE," "PLACE-1," and "PLACE-2" described above. These rules are similar to the macro operators created for the 8-PUZZLE for Korf's macro learning program [Korf 1985].[14] An example of one of the abstract operators is shown in Figure 19. The diagram shows how a sequence

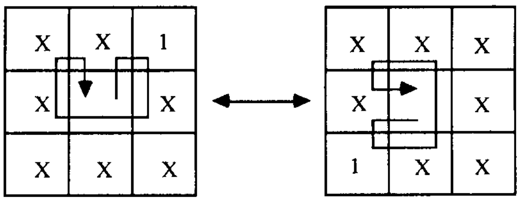


Figure 19. Abstract operator created by SOAR [Laird et al. 1986a].

of rules will guide the one-tile to the correct location whenever (a) the one-tile is at the upper right or lower left corner and (b) the blank is in the center. The "x" marks indicate that the rules apply regardless of the contents of the other cells. As suggested by Figure 19, the chunks apply to a variety of board situations, many of which SOAR has never seen before [Laird et al. 1986a].

In order to evaluate the chunking mechanism in SOAR, it is useful to examine SOAR's behavior before and after chunking takes place. Figure 17 shows how SOAR behaves before chunking takes place. SOAR was forced to resolve three impasses in order to apply the abstract operator "PLACE-BLANK." After building new chunks, SOAR can solve this problem and similar ones without the occurrence of any impasses. This example illustrates that SOAR can create new rules to avoid impasses and the searching that results from impasses. Statistics presented by Laird et al. [1984, 1987] show that chunking reduces the total number of search control decisions that the system must make. Nevertheless, a question arises as to whether the number of search control decisions is an appropriate unit of measurement. Although chunking can reduce the number of control decisions by creating rules used in the elaboration and decision phases, these processes may run more slowly after chunking than before. The difficult work may simply have been moved to a different part of the system. Tambe and Newell [1988] have measured absolute CPU time in SOAR. Their results show that chunking does improve overall performance on some tasks, but degrades others by creating chunks that are expensive to match [Tambe and Newell 1988].

[13] The dependency tracing technique is similar to the methods used in LEX-II and GENESIS. This raises the question of why SOAR does not retrieve definitions of fired production rules and analyze them using a procedure like EGGS. Such an approach might lead to a method of changing constants to variables that avoids problems of overspecialization. The developers of SOAR may have rejected this approach because SOAR is implemented in a variant of OPS5 [Forgy 1981]. Unlike the STRIPS type operators used in GENESIS, the OPS5 productions may be relatively difficult to analyze.

[14] Although SOAR and Korf's system create roughly the same macros, they do not apply macros in the same way. When SOAR forms the macro sequence OP1, . . . , OPN, it applies the operators one at a time. SOAR must make at least one search control decision between applying operators OP(i) and OP(i + 1). To make the decision, SOAR must go through elaboration and decision phases. Korf's system can apply a macro sequence OP1, . . . , OPN as a group without making any search control decisions between applying operators OP(i) and OP(i + 1).

The developers of SOAR have claimed that the power of chunking is enhanced when used in the context of a problem-solving architecture such as SOAR [Laird et al. 1986a], because all parts of the system fall within the scope of the learning mechanism. They argue that the SOAR architecture allows chunking to improve any aspect of the behavior of a problem-solving system. This can enable SOAR to avoid "wandering bottleneck" problems, which have occurred in systems like LEX-II [Mitchell 1983]. This potential capability results from the fact that all decisions in SOAR are made in the same manner, by searching in a problem space. The chunking mechanism can create production rules that summarize the results of search. It follows that the learning system in SOAR has the potential to create rules to guide any of the decisions that the system makes in the course of problem solving. All decisions can be influenced by the chunking mechanism. More practical experience is needed to determine whether this conclusion is borne out in practice.

Although SOAR has been described mainly in terms of chunking, it can also be viewed in other ways (see Figure 3). SOAR can also be viewed as a system that can generalize from a single example. After SOAR solves a goal in one situation, it creates a rule that can solve the same goal when it occurs again in other problem-solving contexts. SOAR is able to generalize across problem-solving contexts by building rules whose conditions only include working memory elements that are necessary for finding the solution of the goal. By omitting the irrelevant working memory elements, SOAR achieves a kind of "implicit generalization" [Laird et al. 1986a]. The learning process in SOAR can also be viewed in terms of reformulation of non-operational concept descriptions. The combination of a goal G and the original production rules may be viewed as a nonoperational specification of the set of problem states in which G can be solved [Rosenbloom and Laird 1986]. The chunking process creates a production rule with conditions that directly test whether the goal G can be solved. The conditions of the new rule may be viewed as an operational description of the same concept.

Initial world model:

INROOM(ROBOT, R1)
INROOM(BOX1, R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R3)
BOX(BOX1)
⋮
$(\forall x, y, z)[\text{CONNECTS}(x, y, z) \Rightarrow \text{CONNECTS}(x, z, y)]$.

Goal formula:

$(\exists x)[\text{BOX}(x) \wedge \text{INROOM}(x, \text{R1})]$.

Figure 20. STRIPS' initial world model [Fikes et al. 1972].

2.3.2 STRIPS (Fikes, Hart, and Nilsson)

STRIPS is a system for building and generalizing "robot plans" [Fikes et al. 1972]. The robot plans are represented as sequences of "STRIPS-type operators." When given a goal to achieve, STRIPS performs a search to find a sequence of operators that transforms the initial state into the goal state. The operator sequences are then combined into chunks called "MACROPS." The sequences are also generalized so that they can be applied to new situations.

STRIPS uses a list of predicate calculus formulas to model the current situation of its world and to describe the goal that the robot plan is intended to achieve. An initial model and a goal are shown in Figure 20. The model describes some interconnecting rooms and the locations of a robot and a box. STRIPS is faced with the goal of getting a box into room R1. The operators that STRIPS can use for this task are shown in Figure 21. Each operator has a set of precondition formulas that must be true in order for the operator to apply to a situation. Before applying an operator, STRIPS uses a resolution theorem prover to verify that the preconditions of the operator are met. Each operator also has an "add list" and a "delete list," which specify the effects of the operator. To apply an operator, STRIPS first instantiates the operator's variables using bindings obtained from the process of proving preconditions. Then

```
GOTHRU(d, r1, r2)
  Precondition:  INROOM(ROBOT, r1) ∧ CONNECTS(d, r1, r2),
  Delete list:   INROOM(ROBOT, r1),
  Add list:      INROOM(ROBOT, r2).

PUSHTHRU(b, d, r1, r2)
  Precondition:  INROOM(ROBOT, r1) ∧ CONNECTS(d, r1, r2)
                 ∧ INROOM(b, r1),
  Delete list:   INROOM(ROBOT, r1)
                 INROOM(b, r1),
  Add list:      INROOM(ROBOT, r2)
                 INROOM(b, r2).
```

Figure 21. Examples of STRIPS operators [Fikes et al. 1972].

1	* INROOM(ROBOT,R1)
	* CONNECTS(D1,R1,R2)
		GOTHRU(D1,R1,R2)	

2	* INROOM(BOX1,R2)
	* CONNECTS(D1,R1,R2)
	* CONNECTS(x,y,z) =>
	CONNECTS(x,z,y)
		* INROOM(ROBOT,R2)	PUSHTHRU(BOX1,D1,R2,R1)

3			INROOM(ROBOT,R1)
			INROOM(BOX1,R1)

	0	1	2

Figure 22. Example of a triangle table [Fikes et al. 1972].

STRIPS deletes any formulas in the current world model that match an item on the delete list. Finally, STRIPS adds all the formulas on the add list to the current model.

After finding a plan to achieve a goal, STRIPS builds a data structure known as a "triangle table." The triangle table describes the structure of the robot plan in a format that is useful for generalizing operator sequences. An example of a triangle table is shown in Figure 22. A procedure for building such a table is shown in Figure 23.[15] The triangle table is useful because it shows how operator preconditions depend on the effects of other operators and on facts from the initial world model. Any fact marked with an asterisk in the table indicates just such a dependency. For example, the marked fact INROOM(ROBOT, R2),

[15] The example table departs slightly from the definition. In column zero of the example, only the "marked" clauses are shown.

0. For an operator sequence of length N, number the rows from 1 to $N + 1$, and number the columns from 0 to N.
1. Place the (i)th operator in the cell at column i, row i.
2. In every cell at column 0, row i $(i = 1, \ldots, N)$, place the facts of the initial model that were true just before the (i)th operator was applied.
3. In the cell at column 0, row $N + 1$, place the facts of the initial model that remained true in the final model.
4. In every cell at column i $(i = 1, \ldots, N - 1)$, row j $(j = i + 1, \ldots, N)$, place the facts added by the (i)th operator that were true just before the (j)th operator was applied.
5. In every cell at column i $(i = 1, \ldots, N)$, row $N + 1$, place the facts added by the (i)th operator that remained true in the final model.
6. Use an * to mark each fact in row j $(j = 1, \ldots, N)$, that was used in the proof of the preconditions of the (j)th operator.

Figure 23. Definition of a triangle table.

1. "Lift" the triangle table.
   (a) Replace each distinct constant in column zero with a distinct variable.
   (b) Replace each clause in column i $(i = 1, \ldots, N)$, with the corresponding clause from the uninstantiated add list of the (i)th operator.
   (c) Rename variables so that clauses from distinct operator applications have variables with distinct names.
2. Rerun proofs of preconditions using isomorphic images of original proofs.
   (a) Each new proof will be supported by the generalized versions of clauses that were marked in the original table.
   (b) Each new proof step performs resolution on pairs of clauses and unification on pairs of literals corresponding to the pairs matched in the same step of the original proof.
   (c) Substitutions generated during unification are applied throughout the entire table.

Figure 24. STRIPS generalization procedure.

in column 1, row 2 of Figure 22, indicates that the precondition of the PUSHTHRU operator depends on a fact added by the GOTHRU operator. Likewise, the presence of the marked fact INROOM(BOX1, R2), in column 0, row 2, indicates that the precondition of PUSHTHRU depends on a fact from the initial model.

There are two main criteria that are used to determine how to generalize the robot plan represented by a triangle table. The first criterion involves maintaining the dependencies between operators. Operator (i) will add a clause supporting operator (j) in the generalized table if and only if the same dependency exists between operators (i) and (j) in the original table. The second criterion requires that the preconditions of operators in the generalized table be provable using the same proofs as used to verify preconditions in the original plan.

STRIPS generalizes operator sequences using the procedure shown in Figure 24. This procedure makes use of both the triangle table and the proofs of operator preconditions that were created when the robot plan was formed. The first step replaces constants with variables leading to an overgeneralized table. The second step constrains the table in accordance with the two aforementioned criteria. The precondition proofs are performed once again. The supporting clauses of the new proofs are the generalized versions of the (marked) supporting clauses of the original proofs. For every step in the original proof that resolved clauses "a" and "b" and unified literals "i" and "j," the new proof resolves the generalized versions of "a" and "b" and unifies the generalized versions of "i" and "j." This technique is similar to EGGS (Figure 7), inasmuch as they both require that the same objects be unified in the generalized proof as in the original proof.

When the STRIPS generalization procedure is used to process the triangle table of Figure 22, it produces the generalized table shown in Figure 25. Several interesting generalizations have been made. The object to be moved from one room to another has been generalized from a BOX to any object. Although the initial and final rooms were identical in the original plan, the room variables are distinct in the generalized plan. STRIPS has also generalized the conditions of applicability of the operator sequence. The marked clauses in the leftmost column of the generalized table indicate the generalized conditions under which the sequence is applicable. Initially, STRIPS only knows that the sequence applies to the initial world model shown in Figure 20. After generalizing the triangle

1	* INROOM(ROBOT,p2)
	* CONNECTS(p3,p2,p5)
		GOTHRU(p3,p2,p5)	

2	* INROOM(p6,p5)
	* CONNECTS(p8,p9,p5)
	* CONNECTS(x,y,z) => CONNECTS(x,z,y)
		* INROOM(ROBOT,p5)	PUSHTHRU(p6,p8,p5,p9)

3			INROOM(ROBOT,p9)
			INROOM(p6,p9)

	0	1	2

Figure 25. Generalized triangle table [Fikes et al. 1972].

table, STRIPS knows the sequence is applicable whenever the conditions in the leftmost column of the generalized table in Figure 25 are met.

An obvious next step would be to create a new STRIPS operator representing the entire generalized operator sequence. The new operator would have the same effect in a single step as the entire sequence of operators used in the original plan. STRIPS does not actually build such a macro operator. STRIPS keeps the generalized triangle table in the form shown in Figure 25 instead. This means that the MACROP cannot be applied in a single step in the course of solving a new planning problem. The operators must be applied one by one. Nevertheless, the table does directly indicate the conditions under which an entire sequence will apply to a problem situation.

STRIPS has been described above in terms of "generalization" and "chunking." It can also be viewed in terms of reformulating nonoperational concept descriptions (see Figure 3). Given an operator sequence $OP_1, \ldots, OP_N$, STRIPS contains all the information needed to determine the condition of applicability of the entire sequence. The information is only present implicitly, embedded in the definitions of the individual operators. One could view the sequence description "$OP_1, \ldots, OP_N$" as a nonoperational description of the condition of application. STRIPS creates an operational description by building the generalized triangle table. The marked clauses in the leftmost column constitute such an operational description of the condition of application.

2.3.3 Similar Work

Anderson's ACT* system is similar to the chunking systems described here [Anderson 1983a, 1983b, 1986]. The ACT* system uses a learning mechanism called "knowledge compilation," which is based on collapsing sequences of production rules into single rules. Each single rule has the same effect as the original sequence from which it was compiled. Anderson describes his ACT* system as a general architecture that underlies all types of human cognition.

Minton and Benjamin have both developed systems that perform chunking in architectures similar to SOAR [Benjamin 1987; Minton 1988a; Minton and Carbonell 1987]. Like SOAR, each system makes a distinction between domain level operators and control rules that guide the application of domain level operators. As in SOAR, the chunks are used to make search control decisions. Minton's PRODIGY system is distinguished by the fact that it learns from failure as well as success. Unlike SOAR, which creates chunks only after successful subgoals, PRODIGY has the additional capability of chunking after subgoal failures. The resulting control rules enable PRODIGY to avoid similar failures in the future.

Many people have investigated methods of forming macro operators outside the context of explanation-based learning. Cheng and Carbonell have investigated methods of building macros with conditional and iterative constructs [Cheng and Carbonell 1986]. Korf developed a method for finding useful macro operators that applies to any problem exhibiting a property called "serial decomposability" [Korf 1985]. Iba investigated heuristics for determining when a sequence of operators will lead to a useful macro operator [Iba 1985]. The REFLECT system of Dawson and Siklossy also had a mechanism for creating macro operators [Dawson and Siklossy 1977].

2.4 EBL = Operationalization

The term *operationalization* may be defined as a process of translating a "nonoperational" expression into an "operational" one. The initial expression might represent a piece of advice, as in Mostow's FOO and BAR programs [Mostow 1981, 1983a], or it might represent a concept, as in Keller's LEXCOP program [Keller 1983]. The initial expression is said to be "nonoperational with respect to an agent" because it is not expressed in terms of data and actions available to the agent [Mostow 1983a]. An operationalizing program faces the task of reformulating the original expression in terms of data and actions that are available to the agent.

As shown in Figure 3, operationalization can be placed into rough correspondence with the EBL generalization processes described previously. The explanation rules used in systems like GENESIS or LEX-II may be viewed as nonoperational specifications of the concepts that these systems learn. The rules "specify" the concepts because they contain all the information needed to construct the learned concepts. The concept specifications are "nonoperational" because the rules only implicitly contain the information. The EBL techniques of GENESIS and LEX-II serve the purpose of making the concepts explicit. Building and analyzing an explanation is similar to the process of translating a nonoperational concept into an operational one. The translation may be said to "explain" how the operational concept description meets the conditions given by the nonoperational concept description.

2.4.1 FOO *and* BAR *(Mostow)*

The FOO and BAR programs were developed by Mostow to investigate the problem of operationalizing "advice." The older FOO program is described in Mostow [1981, 1983b]. BAR was developed as an extension to FOO and is described in Mostow [1983a, 1983c]. The programs were tested mainly in the domain of the card game Hearts. Some additional tests were run in the domain of music composition.

As an example of a nonoperational expression from the hearts domain, consider the advice to "avoid taking points."[16] This advice is considered "nonoperational" because it is not written in terms of actions that a player can perform. The rules of the game do not allow one to refuse to take up the cards at the end of a trick merely because they include point cards. The only actions available to a player are to choose

[16] The phrase "taking points" means winning tricks that contain point cards. In the version of the game described previously, point cards are hearts.

Unfolding concept definitions:

If F is a concept defined as $(\lambda(x_1 \ldots x_N)e)$, then replace the expression $(F\ e_1 \ldots e_N)$ with the result of substituting $e_1 \ldots e_N$ for $x_1 \ldots x_N$ throughout e.

Approximation of a predicate (1):

Given an expression containing (P S), where P is a predicate, replace (P S) with the expression (High (Probability (P S))).

Approximation of a predicate (2):

Given an expression containing (P S), where P is a predicate, replace (P S) with the expression (Possible (P S)), where (Possible (P S)) is true unless (P S) is known to be false.

Figure 26. Problem transformation rules [Mostow 1983].

to play one of the cards from his hand. As another example, consider the advice "Don't lead a card of a suit in which an opponent is void."[17] This advice is not operational because it requires knowing one's opponents' cards. These data are not usually available to a player. Mostow's program can translate the advice to "avoid taking points" into an operational expression. After translation, the advice becomes "play a low card." In this new form, the advice does directly specify an action available to the player and is therefore considered to be operational.

In order to translate a piece of advice, Mostow's programs make use of several types of knowledge. One part of the knowledge base contains a set of domain-independent "problem transformation rules." Each rule has an action component specifying how to rewrite an expression representing some advice as well as conditions governing the applicability of the rule. Examples of such rules are shown in Figure 26. The transformation rules are progressively applied to the initial advice, gradually changing it into a form that meets the requirements of operationality. The knowledge base also contains domain-dependent "concept definitions" like those shown in Figure 27.

[17] A player is said to be "void" in suit if he does not have any cards of that suit in his hand.

The FOO and BAR programs differ in the type of control structure used to choose a sequence of rule applications. FOO relies on a human user to pick an appropriate sequence of transformation rules [Mostow 1983b]. BAR uses means-ends analysis to guide the choice of which rule to apply [Mostow 1983a]. The rule sequences can be quite long, amounting to over 100 rule applications in some cases. Even the BAR program is unable to work without some human guidance.

In order to guide the search process, BAR needs to know which specific parts of an expression are not operational. This is done by annotating each "domain concept" with information that indicates the operationality of the concept [Mostow 1983a]. For example, the concept "point-cards" is marked as being operational since a player always knows which cards are worth points. The "void" predicate is not operational, since a player cannot generally know when an opponent is void in a suit. In general, predicates can be "evaluable" or "not evaluable," functions can be "computable" or "not computable," events can be "controllable" or "not controllable," and constraints are "achievable" or "not achievable." BAR also contains some general knowledge about operationality. For example, there is a rule stating that "a computable function of evaluable arguments is itself evaluable." Another rule says that "an evaluable constraint on a controllable variable is achievable." This knowledge can be used to guide the search process by determining which parts of an expression are nonoperational and need to be transformed.

In order to illustrate the operationalization techniques, consider the following example taken from Cohen and Feigenbaum [1982], which shows how the FOO program operates. FOO is initially given the advice "Avoid taking points," which is represented internally by the expression

(AVOID (TAKE-POINTS ME) (TRICK)).

This expression may be interpreted as saying "Avoid an event in which the player 'me' takes points during the current trick." In order to translate this expression, FOO

```
POINT-CARDS = (LAMBDA ( ) (SET-OF C (CARDS) (HAS-POINTS C))),
       VOID = (LAMBDA (P SUIT)
                  (NOT (EXISTS C (CARDS-IN-HAND P)
                                 (IN-SUIT C SUIT)))),
      AVOID = (LAMBDA (E S) (ACHIEVE (NOT (DURING S E)))),
      TRICK = (LAMBDA ( )
                  (SCENARIO (EACH P (PLAYERS) (PLAY-CARD P))
                            (TAKE-TRICK (TRICK-WINNER)))).
```

Figure 27. Concept definitions [Mostow 1983].

first uses the rule for unfolding concept definitions (Figure 26), along with the definitions of the concepts "avoid" and "trick" (Figure 27). The system subsequently applies several more transformations, including "case analysis," "intersection search," "partial matching," and "simplification" to translate the expression into the form

```
(ACHIEVE (NOT (AND
      (= (TRICK-WINNER ME)
            (TRICK-HAS-POINTS))))).
```

This expression says "Try not to win a trick that contains point cards." After several additional transformations, the final form of the advice is obtained:

```
(ACHIEVE
  (⇒(AND (IN-SUIT-LED
           (CARD-OF ME))
         (POSSIBLE
           (TRICK-HAS-POINTS)))
    (LOW (CARD-OF ME)))).
```

This expression asserts the advice "Play a low card when following suit in a trick that could possibly contain point cards."[18]

This final expression is not exactly equivalent to the original advice. There have been several modifications to the content of the advice as well as the form of the advice. To begin with, the final form of the advice is specialized to a more limited range of situations than the original advice. The final advice only applies in situations when the player is "following suit." The original advice purports to apply to any situation. In addition to specializing the advice, the system was forced to make approximations. One approximation replaced the expression (TRICK-HAS-POINTS) with (POSSIBLE (TRICK-HAS-POINTS)). This was necessary because it is not possible to determine in advance whether a trick will have points. In order to have an operational rule, the system inserts a condition testing whether, based on current information, it is possible for the trick to eventually contain points. Another approximation replaced the requirement of playing a card that will lose the trick with the weaker requirement of playing a low card. Since the player cannot generally determine whether a card will lose a trick, he must use the approximation of playing a low card. This example illustrates the need to sacrifice generality and accuracy in order to translate advice into an operational expression.

The FOO and BAR programs have been described in terms of operationalizing "advice." As suggested by Figure 3, they may also be viewed in terms of operationalizing "concepts" in the following way: Initially the system is given the nonoperational concept description "cards that avoid taking points." This description is translated into the operational form "low cards." FOO and BAR can also be viewed in terms of chunking. After translating the advice, the system may be said to possess a rule of the form "If a card is low, then the card avoids taking points." This rule represents the result of forming a chunk out of the sequence of problem transformation rules used to translate the advice. Although FOO and BAR do not look at examples, they could be modified to implement a process of generalizing from examples. The system could

[18] A player is said to be "following suit" whenever he plays a card in the same suit as the card played by the leader of the current trick.

```
(∀op, s){POSINST(op, s) ⇐ USEFUL(op, s)},
(∀op, s){USEFUL(op, s) ⇐
        [¬SOLVED(s)
         ∧ SOLVABLE(APPLY(op, s))
         ∧ APPLICABLE(op, s)
         ∧ {(∀oop)
            EQUAL(op, oop)
            ∨ ¬APPLICABLE(oop, s)
            ∨ ¬SOLVABLE(APPLY(oop, s))
            ∨ GREATER-COST(APPLY(oop, s), APPLY(op, s))}]},
(∀op, s){SOLVABLE(s) ⇐ SOLVABLE(APPLY(op, s))},
(∀op, s){SOLVABLE(s) ⇐ SOLVED(APPLY(op, s))}.
```

Figure 28. Rules defining the POSINST predicate in LEXCOP [Keller 1983].

be given an example of a "card that avoids taking points." The search for a translation could be constrained by imposing the requirement that the translated advice be capable of predicting the given example. Examples might help the system decide what types of approximations and specializations are appropriate.

2.4.2 LEXCOP and MetaLEX (Keller)

The LEXCOP system [Keller 1983] is closely related to Mostow's operationalizer. Like Mostow's systems, LEXCOP is intended to translate nonoperational expressions into operational ones. The systems differ slightly in the types of expressions they reformulate. Whereas FOO and BAR are designed to reformulate "advice," LEXCOP is explicitly intended to address the problem of reformulating "concept descriptions." LEXCOP takes nonoperational concept descriptions as input and produces operational concept descriptions as output. Keller's system is also distinct from Mostow's because of its criterion for deciding when an expression is operational. In LEXCOP a concept description is operational if it allows instances to be "efficiently" tested for concept membership. LEXCOP uses the same basic methodology as FOO and BAR. The knowledge base contains a set of transformation rules that can rewrite concept descriptions. LEXCOP uses these rules to perform a heuristic search in a space of concept descriptions. Each state is a concept description and the transformation rules are operators of the state space. LEXCOP was worked out on paper but apparently never implemented [Keller 1987a].

Consider the following example from the domain of symbolic integration. A definition of the concept "POSINST(*op*, *s*)" is shown in Figure 28. This definition asserts that a state "*s*" is a positive instance if applying "*op*" to "*s*" leads to a state along a minimum-cost solution path. In this form the concept description is considered to be "nonoperational." For example, in order to test a state "*s*" for membership in the concept POSINST(OP1, s), it may be necessary to build a large search tree. LEXCOP attempts to reformulate this concept description into something that can be tested more efficiently. Given the nonoperational description POSINST(OP1, s), LEXCOP could produce the description shown in Figure 29. This new concept description can be tested more efficiently because it is written as a pattern match using the generalization language of LEX [Mitchell et al. 1983a]. Notice that the translated description is a specialization of the original concept description. Like Mostow's systems, LEXCOP is forced to sacrifice generality in order to make an expression more operational. In order that the new concept description be useful in a variety of situations, LEXCOP would have to create a conjunction of several alternate specializations of the original concept description.

Some of the transformation rules used in LEXCOP are shown in Figure 30, taken from [Keller 1983]. The rules are divided into three main types. The "concept-

$$(\forall s)\{\text{POSINST}(\text{OP1}, s) \Leftarrow \text{MATCH}(\langle\text{function}\rangle \int \sin(x)\, dx, s)\}.$$

Figure 29. Translated concept description [Keller 1983].

Concept-preserving transforms:
1. Expand definition of a predicate.
2. Constraint back-propagation.
3. Enumerate the values of a universal variable.

Concept-specializing transforms:
1. Add a conjunct to an expression.
2. Delete a disjunct from an expression.
3. Instantiate a universal variable.

Concept-generalizing transforms:
1. Add a disjunct to an expression.
2. Delete a conjunct from an expression.

Figure 30. Transformation rules in LEXCOP [Keller 1983].

preserving transformations" rewrite concepts without changing their meaning. The "concept-specializing" and "concept-generalizing" transformations make concepts more specialized and more generalized, respectively. A concept-specializing rule creates a new expression representing sufficient conditions for concept membership. A concept-generalizing rule produces a new expression representing necessary conditions for concept membership. The sequences of transformations used in LEXCOP correspond closely to the explanation trees used in LEX-II [Mitchell 1983]; however, the explanation trees of LEX-II are built from concept-preserving and concept-specializing transformations only. This explains why LEX-II creates generalizations that represent sufficient, but not necessary, conditions for concept membership. Unlike the LEX-II system, LEXCOP would arrive at the translated concept description without making use of any training examples.

Keller has developed a new system called MetaLEX, which builds on the ideas of LEXCOP [Keller 1987a]. MetaLEX is intended to show how learning systems can exploit explicit representations of **contextual knowledge**, that is, knowledge of the context in which learning takes place. Keller defines "contextual knowledge" to include (1) a description of the performance program to be improved by learning and (2) a specification of performance objectives, among other things.

Keller argues that contextual knowledge is useful for several purposes. For example, he outlines a method by which a learning system can utilize contextual knowledge to automatically formulate its own learning tasks. By analyzing the algorithm used in the performance program, a learning system could formulate a plan to improve performance by inserting a concept membership test at some location in the performance program. The plan would initially describe the concept in nonoperational terms. The learning system would then be faced with the task of translating the concept into an operational form. By formulating new learning tasks to attack bottlenecks as they move around in the system, this process may provide a solution to the "wandering bottleneck" problem.

Unlike FOO, BAR, and LEXCOP, MetaLEX uses empirical information. In particular, MetaLEX collects data measuring the CPU time expended in evaluating various parts of a concept description. By indicating which parts of an expression are the least operational and most in need of reformulation, these empirical data help to guide the search for an operational concept description. MetaLEX also collects data to help determine when a concept can be safely approximated. MetaLEX measures the impact of various approximations on overall efficiency and accuracy. By comparing these to the system's performance objectives (i.e., contextual knowledge), MetaLEX can determine when approximations are worthwhile.

2.4.3 Similar Work

Techniques for operationalization have not been studied extensively in the field of machine learning. Some automatic programming methods can be viewed in terms of operationalization. The transformational

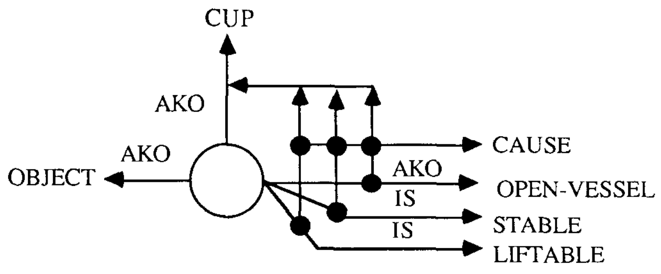


Figure 31. Functional definition of a cup [Winston et al. 1983].

implementation methodology developed by Balzer is a case in point [Balzer et al. 1976]. This technique takes a (nonoperational) program specification as input. A series of correctness-preserving transformations are then applied to the specification, gradually refining it into an executable (operational) program. This method has been used by Swartout to build knowledge-based expert systems for which human-oriented explanations can easily be generated [Swartout 1983]. A survey of program transformation systems is found in Partsch and Steinbrüggen [1983]. The relation between EBL and program transformation is discussed in Prieditis [1988a].

2.5 EBL = Justified Analogy

Techniques for performing "justified" analogical reasoning are discussed in this section. Traditional methods of reasoning by analogy require making a guess about what information should be transferred from a remembered analogous situation to a new situation. The "justified" version of analogy tries to avoid guessing. One approach to justified analogy involves mapping sequences of "inference rules," or "explanations," from analogs to target examples. The inference rules might encode "causal relations" as in Winston et al. [1983], Kedar-Cabelli [1985], and Gentner [1983], or they might represent problem-solving "derivation" steps as in Carbonell [1986] (see Figure 3). Since the inference rules contain their conditions of applicability, the system needs only to verify that the mapped rules apply to the new situation in order to avoid making guesses. This suggests that **explanation-based analogy** (EBA) would be a reasonable name for these techniques.

2.5.1 ANALOGY (Winston)

Winston and his co-workers have developed the ANALOGY system [Winston et al. 1983]. This program is intended to learn "physical" or "structural" descriptions of objects. The program is given "functional definitions" of objects as input. By finding analogies between "precedents" and "practice examples," ANALOGY transforms the functional definition into a physical or structural description.

The ANALOGY program is described using the example of a drinking cup. The input to the system is a functional definition of a cup, shown in Figure 31. This definition gives three conditions that must be met in order that an object function as a drinking cup. The object must be a "stable, liftable, open vessel." These conditions are considered to be functional specifications but not physical or structural properties. A variety of physically different objects could fulfill these three functional criteria. In addition to a functional definition, the system is also given an example of a cup, shown in Figure 32. ANALOGY so provided with a set of precedents that are used to reason by analogy. These precedents include descriptions of objects such as bricks, suitcases, and bowls, which are useful for establishing the connection between physical properties and functional specifications.

ANALOGY begins by trying to confirm that the example is indeed a cup. The functional definition network is retrieved and superimposed on the example network.

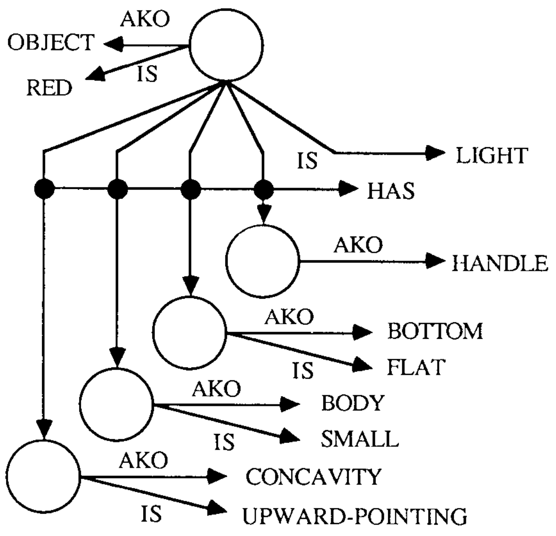


Figure 32. Example of a cup [Winston et al. 1983].

Next the system tries to establish each of the three criteria in the definition; that is, the program must show that the example is a "stable, liftable, open vessel." Each condition can be established either by verifying that the condition appears directly in the description of the example or by reasoning from a precedent. The suitcase precedent is used to show that the example is liftable. The description of the suitcase precedent contains a causal chain. This chain has two steps asserting that (1) "the suitcase is liftable because it is light and graspable" and (2) "the suitcase is graspable because it has a handle." In order to use the chain, ANALOGY determines a correspondence between parts of the cup example and parts of the suitcase precedent, using a method called "importance-dominated matching" [Winston 1982]. While transferring the chain, the program tests whether the antecedents of the chain are found in the example. In this case the cup example does in fact contain the "light" and "handle" relations. This means the condition of being "liftable" is successfully established. In a similar manner, ANALOGY uses the brick precedent to show that the example is stable and the bowl precedent to show that the example is an open vessel. The final version of the example network is shown in Figure 33. This diagram shows all the causal chains transferred from the precedents to the cup example.

After establishing the example to be a cup, ANALOGY creates a general rule. The rule is intended to summarize the set of physical properties that enabled the example to function as a cup. An English paraphrase of the new rule is shown in Figure 34. The "IF" part of this rule was built from the antecedents of the causal chains transferred from precedents. The "THEN" part asserts an object to be a cup. The "UNLESS" conditions correspond to the intermediate nodes of the transferred causal chains. These conditions are included because the causal connections are not considered to be infallible. For example, the causal link asserting that "an object is graspable if it has a handle" might be wrong in some cases. By adding the "UNLESS" condition, the rule is understood to mean "an object is graspable if it has a handle, unless there is some reason to believe otherwise."

A question arises regarding whether the precedents are really necessary in the ANALOGY system. According to Winston, "The precedents are essential for otherwise there would be no way to know which aspects of the example are relevant" [Winston et al. 1983, p. 433]. The precedents

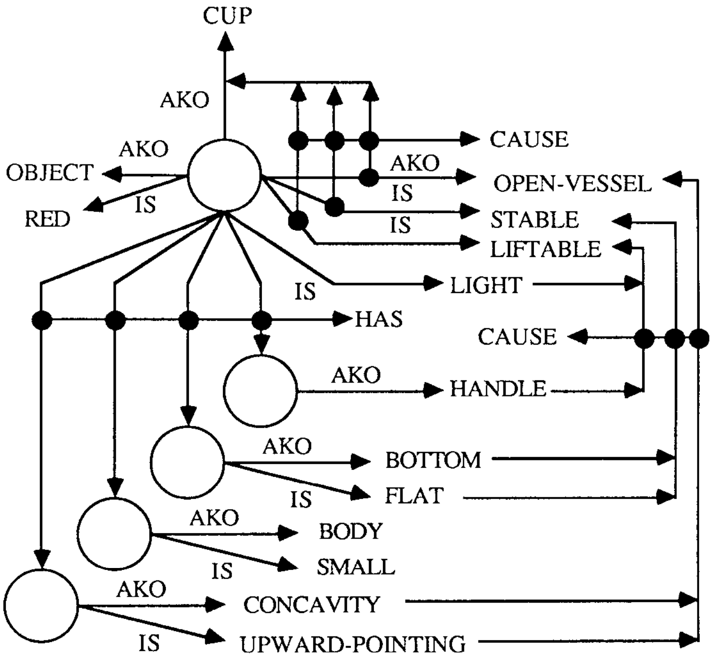


Figure 33. Final version of example network [Winston et al. 1983].

IF:	AN OBJECT IS LIGHT AND HAS A HANDLE, A FLAT BOTTOM AND AN UPWARD POINTING CONCAVITY,
THEN:	THE OBJECT IS A CUP,
UNLESS:	THE OBJECT IS NOT STABLE, OR NOT LIFTABLE, OR NOT AN OPEN VESSEL, OR NOT GRASPABLE.

Figure 34. Rule extracted from network.

might appear to be necessary because they contain causal information in the form of links between causes and effects. ANALOGY may be said to possess an "extensional theory" of causes and effects in the form of precedents. This can be contrasted with an "intensional theory" in the form of general rules connecting causes and effects [Mitchell et al. 1986]. Nevertheless, Winston's database of precedents is really an intensional theory in disguise. ANALOGY has the ability to extract causal relations from precedents and transfer them to new situations. This implies it can determine which conditions must hold for a causal link to be in effect. As observed by Mitchell, the ANALOGY program implicitly assumes a causal link such as "FEATURE1(A) $\rightarrow$ FEATURE2(A)" is supported by a general rule of the form "$(\forall x)\{\text{FEATURE1}(x) \Rightarrow \text{FEATURE2}(x)\}$" [Mitchell et al. 1986]. If a database of rules were created by extracting causal links from the precedents, the result would be a program looking more like GENESIS or LEX-II. There may be a reason for storing causal rules in the context of precedents. The causal rules may be faulty. When contradicted by future information, they will need revision. The precedents might help determine how to revise faulty rules.

Winston's ANALOGY program can also be viewed in terms of generalization, chunking, and operationalization (see Figure 3). The rule in Figure 34 can be taken as a generalization of the single example of a cup, which was provided to the system. The rule may also be seen as an operationalization of the functional definition of a cup. The original definition of a cup in Figure 31 may be considered to be "nonoperational" because it describes a cup in functional terms. The final rule in Figure 34 is operational because it describes cups in physical or structural terms. Winston's program also performs chunking. Three causal chains are taken from three precedents, the suitcase, the brick, and the bowl, and are spliced together to build an explanation of the cup example. The explanation is then collapsed into a single rule representing a chunk.

2.5.2 Derivational Analogy (Carbonell)

Derivational Analogy (DA) was developed by Carbonell to investigate analogical reasoning in the context of problem solving [Carbonell 1983a, 1986]. The DA technique solves a new problem by making use of a solution derivation that was generated while solving a previous problem. The new problem is solved by recreating sequences of decisions and justifications for decisions that were used to solve a precedent problem. Carbonell uses derivations in a way similar to the manner in which Winston uses causal networks. Carbonell proposes transferring derivations between examples, whereas Winston proposes transferring causal networks. Derivations and causal networks are both types of dependencies or justifications. Inasmuch as DA involves transferring justifications from a precedent to a new situation, it may be seen as a type of justified analogical reasoning.

The DA method was originally developed to remedy a limitation of earlier work on analogy in problem solving. Carbonell's earlier work involved solving new problems by directly modifying solutions to previously solved problems [Carbonell 1983b]. For example, one might try to write a sorting program by directly modifying the code used in a previous sorting program. The difficulty can be illustrated by considering the following problem from Carbonell [1986]: Suppose one wanted to write a LISP sorting program, and one had already written a Pascal program implementing quicksort. The approach of directly modifying the Pascal program would either fail completely or lead to a poor LISP program. This would happen because a good LISP implementation of quicksort would look quite different from the Pascal program owing to differences in the structures of these languages. Nevertheless, the LISP and Pascal programs might share the same underlying design strategy. They could both use a divide and conquer approach manifested in terms of partitioning sets. This strategic information is ignored by an analogy process that directly transforms the code of one program into the code of another. DA avoids this problem since it does not try to directly transform one solution into another. The DA method transfers information at the level of "derivations" rather than "solutions." DA would solve the sorting problem by transforming the derivation of the Pascal program into a derivation of a LISP program.

Carbonell gives a detailed specification of the sorts of information that should be contained in a derivation [Carbonell 1983a, 1986]. A derivation is supposed to include the "hierarchical goal structure" used to generate the solution. The goal structure is represented in terms of the "sequence of decisions" made while solving a problem. For each decision, the derivation should list the alternative that was chosen as well as those that were considered, but not chosen. The record of a decision should include the reasons for the decision (i.e., the derivation might record an explanation of the decision along with dependency links to aspects of the problem specification and dependency links to general knowledge). The derivation should also indicate how each decision depends on prior decisions and influences subsequent decisions. Finally, the derivation should record the initial segments of any dead-end paths that were explored, along with reasons the paths appeared promising and reasons the paths ultimately failed.

In order to use the DA method to solve a problem, it is necessary to find prior problem situations that are analogous to the current situation. DA begins solving a problem by using general techniques, for example, application of weak methods or instantiation of a general problem-solving schema like divide and conquer [Carbonell 1986]. A trace is maintained to record these initial stages of the problem-solving process. Appropriate analogous problems are found by matching the initial analysis trace of the current problem with the initial analysis of previous problems.

After finding an analogous problem, the derivation of the analogous problems' solution is retrieved and applied to the new situation. A derivation may be transferred to a new problem in the following way: The system must follow the sequence of decisions in the derivation and reconsider each one in the context of the current problem. In order to reconsider each decision, the system must examine the reasons for the decision. This can be done by examining the dependency links to the previous problem situation and to general knowledge. If the relevant aspects of the problem specification are the same and the general knowledge applies to the new situation, then the same decision can be made. Otherwise, the system must reconsider the decision. Carbonell actually provides a more detailed description of how to transfer a derivation from one problem to another [Carbonell 1986].

2.5.3 Analogy versus Generalization

The explanation-based versions of analogy and generalization differ mainly on the issue of schema formation. Systems like GENESIS and SOAR are naturally viewed as generalizers because they convert derivations (explanations, operator sequences) into schemata (chunks). The schemata represent compiled versions of the derivations and need only to be instantiated to apply to new problems. The process of schema instantiation solves a new problem in a single step, bypassing all the intermediate steps of the derivation. In contrast, Carbonell's DA method is more naturally viewed in terms of analogy because it does not convert a derivation into a schema, but rather keeps the derivation in its original form. In order to solve a new problem, DA must pass through all the steps in the original derivation, possibly modifying them to some degree.

Each approach has advantages. When a new problem actually matches an existing schema, the process of schema instantiation is usually more efficient than replaying an entire derivation. Schemata suffer from the disadvantage of not being immediately useful when a new problem falls outside their scope. Derivational analogy does not suffer from this problem. If one assumes that DA can modify derivation steps so the derivation can apply to a new problem, then the original derivation does not have a fixed range of application.

2.5.4 Similar Work

The EBA methods discussed in this section are similar to other recent research in analogical reasoning. In particular, they are similar to Gentner's "structure-mapping" theory of analogy [Gentner 1983]. This theory involves using a principle called "systematicity" to determine what information should be mapped from the analog to the target example. According to the systematicity principle, analogy processes should transfer "systems of relations." A system of relations involves "first-order" relations that are governed by "higher order" relations. Causal relations are one type of higher order relation. The systematicity criterion often leads to transferring networks of causal relations from one example to another. The causal nets can be interpreted as explanations. For this reason the systematicity principle often results in transferring explanations from the analog to the target, just as in explanation-based analogy. Despite this similarity, structure mapping is different from EBA in one crucial respect. Although the structure-mapping theory often leads to transfer of explanations, it does not actually require that all analogical inferences be logically sound.

A method of justified analogical reasoning, called "Purpose-Directed Analogy" (PDA), has been proposed by Kedar-Cabelli [1985]. PDA is intended to address the question of deciding which causal network should be transferred from the analog to the target, in cases when the analog contains many possible causal networks. Kedar-Cabelli argues that the methods of Winston and Gentner are not able to operate unless the relevant network is specified in advance. PDA tries to avoid this limitation by using the "purpose of the analogy" to select the relevant network from among many.

Derivational analogy has been applied to the logic circuit design domain in several systems. For example, the REDESIGN system [Mitchell et al. 1983b; Steinberg and Mitchell 1985] serves as an assistant to a human for the purpose of designing new circuits by analogy with existing ones. REDESIGN combines causal reasoning about circuit behavior with knowledge about the design plan of the original circuit in order to focus attention on the parts that must be modified. Other applications of derivational analogy to circuit design include the BOGART system [Mostow and Barley 1987] and the ARGO system [Huhns and Acosta 1987].

Mostow has investigated the applicability of derivational analogy to design problems in general, including both circuit design and program generation [Mostow 1986]. He has examined some difficulties that arise in the course of attempting to replay derivations. This study has led him to propose criteria about the types of information that should be included in derivations. Mostow has also done a comparative analysis of several derivational analogy systems in design domains [Mostow 1987a].

An entirely different approach to justified analogy has been developed by Davies and Russell [1987]. Their technique involves utilizing "determinations," for example, a rule asserting that "the value of feature A determines the value of feature B." A system in possession of determinations can make logically sound inferences from precedents to new examples. Other knowledge-intensive approaches to analogical reasoning are discussed in Prieditis [1988b].

2.6 Additional EBL Research

Several additional research projects are related to explanation-based learning in the sense that they use explanations to guide learning; however, they do not fit neatly into any of the four categories: generalization, chunking, operationalization, or analogy. Systems developed by Silver and by Schank fall into this group. Silver [1986a] has built a program called LP, which learns heuristics for solving algebraic equations. LP uses an analytic learning technique called "precondition analysis" (PA). The PA method is used to infer the strategic purpose of an operator, when the LP system sees it used within a sequence of operators. Suppose that the two operators, P_i and P_{i+1}, appear within the sequence, $P_1, \ldots, P_{i-1}, P_i, P_{i+1}, \ldots, P_N$. The PA method will assume that P_i was used to achieve some preconditions of P_{i+1}. Suppose that A is a set containing all the preconditions of operator P_{i+1} and that B is a set containing members of A that are true before P_i was applied. The set difference, $A - B$, represents those preconditions of P_{i+1} that were brought about by the operator P_i. PA would then infer that these conditions are the "strategic purpose" of P_i. After learning the purpose of an operator, LP would use the information as a search control heuristic in future problem solving. Precondition analysis is related to EBL methods in two ways. It can learn from a single observation of an operator sequence applied to an algebra problem. It also relies on background knowledge about the preconditions of operators. Precondition analysis differs somewhat from analytical techniques like constraint-back propagation (CBP) and EGGS. PA can be applied to operators that are ill behaved in certain ways that would cause these methods to fail [Silver 1986b].

Schank and co-workers have been working on a theory of learning and memory that is similar to EBL. Schank envisions a role for explanations in learning; however, he uses explanations in a somewhat different way than the EBL systems described above. He has proposed a theory called "failure-driven memory" (FDM) based on the idea that learning is possible whenever a person encounters a failure of expectations [Schank 1982]. In the course of attempting to explain the failure, a person is reminded of previous episodes that can be understood using the same explanation. Such reminding is possible if memory is indexed in terms of "patterns of explanation." Schank has proposed a typology of standard explanation patterns [Schank 1987]. Hammond has used the FDM method in his CHEF program [Hammond 1986, 1987].

Schank's FDM theory can be compared with EBL in the following way: According to FDM, if event A causes one to be reminded of event B, then A and B share a common explanation. In the context of EBL, if A and B are instances of a single generalization, then they can both be understood using the same explanation. Owing to the emphasis that Schank places on case-based reasoning, his work bears an especially strong resemblance to explanation-based analogy.

3. FORMALIZATIONS OF EXPLANATION-BASED LEARNING

3.1 Mitchell's EBG Formalism

A formalism called **explanation-based generalization** (EBG)[19] has been proposed by Mitchell et al. [1986]. EBG attempts to capture the essential elements of most explanation-based learning systems that have been proposed. EBG is similar in spirit to Mitchell and Utgoff's LEX-II system; however, it uses a more uniform set of methods and is cast in a form that is more clearly applicable to other domains. Mitchell describes the EBG framework as a "domain independent method . . . for using domain dependent knowledge to guide generalization" [Mitchell et al. 1986, p. 49].

The EBG formalism consists of two parts called the "EBG Problem" and the "EBG Method." A formal specification of the problem is shown in Figure 35. The EBG problem is defined in terms of four parameters that are necessary for all EBG systems. The "goal concept" represents the objective of the learning program. This parameter provides a nonoperational specification of the concept that the system will attempt to learn. In Mitchell's presentation of EBG, the goal concept is represented as

Given:
(1) Goal concept,
(2) Training example,
(3) Domain theory,
(4) Operationality criterion,

Find:
A new concept description that is
(a) a generalization of the training example,
(b) a sufficient condition for the goal concept, and
(c) that satisfies the operationality criterion.

Figure 35. The EBG problem.

an atomic predicate calculus formula, possibly containing free variables (e.g., variables "*s*" and "*obj*" in POSINST(OP3, *s*) and CUP(*obj*)). The "operationality criterion" specifies the types of concept descriptions that are considered to be operational. Mitchell represents the criterion as a list of predicates that are observable or easily evaluable. A concept description is considered operational if and only if it is expressed entirely in terms of predicates from this list. The "training example" is a description of an object that is an instance of the goal concept. The training example parameter is described in operational terms, that is, using predicates from the list of operational predicates. Finally, the "domain theory" parameter is a set of rules describing the domain from which the example and goal concept are drawn. The rules must be capable of proving that the training example meets the conditions for being an instance of the goal concept. In Mitchell's presentation, the domain theory is represented as a set of Horn clauses.

The EBG system is charged with the task of reformulating the goal concept into an expression that meets the operationality criterion. The new concept description need not be exactly equivalent to the original goal concept, so long as it is both (1) a specialization of the goal concept and (2) a generalization of the training example. In order to create such a concept description, the EBG system uses a two-step process similar to the ones described above for GENESIS and LEX-II. First the system uses the domain theory to build an explanation tree proving that the training example satisfies the goal concept defini-

[19] In the remaining sections, the term *EBG* shall refer to Mitchell's specific formalism, whereas *EBL* refers to any explanation-based learning system.

Generalization:

Training example → Operational concept description.

Chunking:

Domain theory → Concept membership test rule.

Operationalization:

Goal concept → Operational concept description.

Analogy:

Training and test examples → Test example classification.

Figure 36. Four interpretations of EBG.

tion. Then the system "regresses" the goal concept formula through the explanation tree to obtain a generalized operational concept description at the leaves. For this step, EBG uses a procedure called **modified goal regression** (MGR) [Mitchell et al. 1986].[20] MGR is a modified version of the goal regression technique described in Nilsson [1980] and Waldinger [1977]. MGR fulfills conceptually the same function as Dijkstra's method of calculating weakest preconditions [Dijkstra 1976], Utgoff's constraint-back propagation [Utgoff 1986], STRIPS' method of generalizing resolution proofs [Fikes et al. 1972], and Mooney and DeJong's EGGS procedure [DeJong and Mooney 1986; Mooney and Bennett 1986].

Mitchell's EBG formalism is valuable for the conceptual clarity it provides. It is especially helpful in making the "goal concept" and "operationality criterion" into explicit parameters. In previously existing EBL systems, these two parameters were present only implicitly. By making them into explicit parameters, the EBG formalism raises the question of how they may be obtained. As outlined by Keller [1987a], these parameters might be generated automatically by a learning program in possession of "contextual knowledge" describing the task and internal architecture of the performance element.

The EBG formalism is also useful for clarifying the relation between generalization, chunking, operationalization, and analogy. Figure 36 suggests how EBG can be interpreted in terms of each of these processes. Each interpretation involves emphasizing one input and one output and ignoring the others. If the training example is the input and the operational concept description is the output, EBG looks like generalization. In order for EBG to look like chunking, the domain theory is taken as the input. The output is a concept membership test rule of the form "if OCD, then GC," where OCD is the operational concept description and GC is the goal concept. EBG looks like operationalization if the input is the nonoperational goal concept and the output is the operational concept description. In order for EBG to look like analogy, the system would be given the training example and a "test" example as inputs. The output would be the classification of the "test" example as a member or nonmember of the goal concept.

3.2 Other Formalizations

DeJong has recently presented a detailed critique of EBG, covering a number of specific areas in which he claims EBG is deficient [DeJong and Mooney 1986]. Among other things, DeJong argues that EBG suffers from problems of undergeneralization. He points out that EBG cannot generalize the predicates appearing in domain theory rules and cannot generalize the structure of the explanation itself. DeJong also discusses other problems with EBG. He claims that the operationality criterion used in EBG is deficient. He also argues that the

[20] The version of MGR in Mitchell et al. [1986] contains an error that was pointed out and corrected in DeJong and Mooney [1986].

EBG generalization procedure fails to take adequate account of the source of the explanation, that is, whether the explanation is built by the system or provided by a human expert. According to DeJong, many of these problems can be solved by organizing the system's knowledge base in terms of a hierarchy of schemata. He presents his own formalism as an alternative to EBG [DeJong and Mooney 1986].

Several other authors have attempted to formally describe the relations among EBL programs. Laird and Rosenbloom [1986] examine the relation between EBG and SOAR. Mostow [1987b] compares the types of knowledge used in several EBL programs by viewing each as performing a search for operational concept descriptions. EBG is shown to be equivalent to "partial evaluation" of logic programs in [Prieditis 1988a] and [Van Harmelen and Bundy 1988]. A domain-independent definition of the term *explanation structure* is presented by Mooney and Bennett [1986]. An early attempt to formalize EBL was made by Minton [1984]. A formalization of explanation-based analogy is presented by Kedar-Cabelli [1985].

4. AN EVALUATION OF EBL

The EBG formalism clarifies a number of outstanding issues in the field of explanation-based learning. Mitchell's formalism draws attention to the fact that an EBG system must be provided with a "domain theory" and a "goal concept" at the outset, before learning can occur. Several questions are suggested by this fact:

- Are training examples necessary for EBG systems?
- Do EBG systems only learn things already contained in the domain theory?
- In what sense can EBG be said to improve an intelligent system?

The first question results from the following observation: If an EBG system possesses a domain theory capable of explaining an example, the same theory might be sufficient for generating the example in the first place. If the system can generate its own example, the training example parameter appears not to be necessary.

In some domains the ability to explain an example is not equivalent to the ability to generate an example in the first place. To illustrate, consider the 8-QUEENS problem. Suppose an EBG system were given the goal concept "mutually nonattacking positions of 8 queens." Given a theory about how queens can attack, a system could easily verify that a solution satisfies the goal concept. Nevertheless, it is much more difficult to find a solution than to verify the correctness of a solution provided by a teacher. This argument applies to the whole class of NP-complete problems, of which the *N*-QUEENS problem is an instance. The NP-complete problems all have the property that solutions are easy to verify but difficult to find. For such problems an example solution provided by a teacher can be very useful.

Mitchell's EBG method might actually be modified to operate without training examples. This would require omitting the step that involves explaining how the example satisfies the goal concept. The following "explanation step" would be used instead: The system would find any explanation tree that has the goal concept at the root and only operational statements at the leaves. The modified explanation process would be permitted to use any operational predicate as an assumption in the explanation. The resulting explanation process might be more time consuming than that occurring if an example were being explained. Some search control techniques that would be useful in the presence of a training example would not apply to the modified explanation process. If Mitchell's EBG method were modified in this way, the result would look very much like Keller's LEXCOP system.

Training examples provided by a teacher may be useful, even if an EBG system could generate its own, or operate without them. A human teacher may be able to select examples that are "typical" of those likely to be encountered by a system in the course of future problem solving. In order to see why typical examples are important, consider that EBG normally does not produce an exact reformulation of the goal concept. It usually creates a specialization of the original goal concept instead. If EBG were applied to an atypical example, it might produce a concept specialization that applies to few typical examples. On the other hand, were a teacher to provide a typical example to the EBG system, the resulting operational concept description is guaranteed to apply to at least one typical example. If the teacher chooses the example carefully, the resulting concept description might be expected to have wide application.[21]

Considering that EBG creates concept recognition rules that are deducible from the initial domain theory, one is led to ask whether EBG systems can learn anything that they do not already know. One answer to this question is provided by Dietterich's definition of "**knowledge-level learning**" [Dietterich 1986]. A system is said to perform knowledge-level learning only when there is a change in the "deductive closure" of its domain theory. The deductive closure of a set of axioms is defined to include the axioms themselves, plus all facts derivable from the axioms using an arbitrary number of inference steps. The concept membership test rules created by EBG are all contained in the deductive closure of the initial domain theory. Therefore, EBG does not change the deductive closure of a knowledge base and does not perform knowledge-level learning. The same criticism applies to most, if not all, of the EBL systems described above.

Russell has proposed an alternate form of justified generalization that does perform knowledge-level learning [Russell 1986]. He suggests that a generalization system's background knowledge may contain rules describing "high-level regularities." One type of high-level regularity is a "determination" rule of the type mentioned above in the context of Russell's justified version of analogy (Section 2.5). Given a determination rule plus a training example, a system can logically deduce a generalization. Nevertheless, the generalization is not derivable from the determination rule alone, in the absence of the training example.

Since EBG does not perform knowledge-level learning, one is led to ask what type of learning EBG actually does perform. Subramanian and Smith [1988] have proposed the idea of "limited knowledge-level learning" to address this issue. Their approach is based on the observation that the complete deductive closure is not generally obtainable in real systems with limited computational resources. One may therefore consider the "knowledge of a system" to include all facts derivable under some limited inference procedure, for example, one using at most a fixed number of inference steps. From this point of view, EBG does change the knowledge contained in a system, by creating chunked rules that enable some theorems to be proved in fewer steps. Thus, EBG may be said to perform limited knowledge-level learning.

The preceding observations suggest that EBG is useful chiefly for the purpose of improving the efficiency of an inference process. It is clear that EBG can produce new rules enabling shorter proofs of some theorems. Nevertheless, this may not improve the overall efficiency of a performance program. As observed by Minton [1985] and Fikes et al. [1972], an EBL system might create rules that are rarely useful. The useless rules consume storage space. Time efficiency may also be degraded if the system is forced to waste time attempting to apply the useless rules. As argued by Minton [1988b], indexing methods or parallel processing may mitigate the problem, but cannot eliminate it entirely.

Several investigators have attempted to empirically measure the efficiency change resulting from EBL. Minton has performed tests showing that EBL can improve or degrade performance, depending on whether the technique is applied in a selective or an uncontrolled fashion. His measurements show that uncontrolled chunk

[21] This may explain a difficulty encountered by LEX-II. LEX-II does make use of training examples, but the examples are generated internally by the problem-generator and problem-solver modules. The learning process in LEX-II was ultimately limited, in part, by the problem generator's inability to choose good examples [Mitchell 1983].

formation can degrade performance [Minton 1985, 1988a, 1988b]. He also shows that EBL can improve performance, provided that heuristics are used to decide when to create and retain chunks [Minton 1988b] (see Section 5.1). Similar empirical results have been reported by Tambe and Newell [1988] and Markovitch and Scott [1988]. Tambe and Newell have performed measurements of CPU time in SOAR. They show that chunking does improve overall performance on some tasks, but degrades others by creating chunks that are expensive to match (see Section 2.3.1). Markovitch and Scott have done an empirical study showing that performance degrades if too many macros are formed, but that improvements result from either random or selective deleting of excess macros. Additional empirical tests of EBL are described by O'Rorke [1987] and Prieditis and Mostow [1987].

This section has provided only partial answers to the three questions listed above. To some extent these questions remain unanswered. By focusing attention on these issues, Mitchell's formalism raises questions about the value of explanation-based learning. EBG may turn out to be a case of premature formalization. Future research may demonstrate that the real value of explanation-based learning comes in situations that do not fit into the EBG framework. In particular, explanation-based learning may be most useful in cases where the initial domain theory is defective.

5. CURRENT AND FUTURE EBL RESEARCH

A large number of problems in the EBL field remain unsolved. Directions for future work may be categorized according to the position they take with respect to a criticism of EBG described in the last section, that is, that EBG does not perform knowledge-level learning. One category can be called "EBL and theory reformulation." This approach would accept the view that EBL can only reformulate an existing domain theory without changing its substantive content. EBL is nevertheless considered worthwhile because it reformulates the theory to make it more "useful." This is the point of view taken in Mitchell's EBG formalism. Even if one works within the EBG paradigm, many problems remain unsolved. Most of this research would examine the question of what makes a theory useful and how EBL should be practiced to guarantee a maximally useful reformulated theory. Another category can be called "EBL and theory revision." This research would seek to take EBL beyond Mitchell's EBG formalism to include methods that change the substantive content of an initial domain theory. The "imperfect theory problem," described below, is included in this category. An additional line of research would seek to develop "integrated" learning methods that combine the analytical approach of EBL with empirical learning methods. To some extent this category cuts across the others. As suggested below, integrated methods may be useful in the context of both theory reformulation and theory revision.

5.1 EBL and Theory Reformulation

5.1.1 Optimization of Reformulated Theories

Researchers who continue to work in the "theory reformulation" paradigm must develop methods to guarantee that EBL really does produce a reformulated theory that is more useful than the initial theory. As discussed in the last section, EBL may or may not improve overall performance, depending on whether the technique is applied selectively or indiscriminately. This problem can be factored into several parts. One part involves deciding which examples should be processed by EBL to form general schemata. Another part involves deciding which schemata should be retained in memory over the lifetime of a program.

5.1.2 When Is Schema Formation Warranted?

In order to avoid creating useless schemata, an EBL program might try to predict in advance whether an example will generalize into a useful schema. A number of investigators have proposed heuristics for deciding when to create schemata. DeJong [1986] proposed a series of questions that a learning system should ask when deciding whether to generalize an observed example plan. His questions are directed mainly toward determining whether the solved goal will occur often and whether the generalized plan will apply in a variety of situations. Schank's theory of "failure-driven learning" suggests paying attention to anomalous situations [Schank 1982]. Lebowitz [1986a] has proposed "interestingness" as a criterion for determining when learning should occur.

Heuristics applying to search spaces with evaluation functions were studied by Iba [1985] and Minton [1985]. Iba's heuristic suggests forming a chunk whenever an operator sequence is found to span two peaks of an evaluation function. Minton's criterion says that generalization should occur when a tough problem is solved in a surprisingly simple way, for example, when a heuristic evaluation function grossly overestimates the cost of a solution. His PRODIGY program also uses "training example selection heuristics" to select those examples that appear likely to result in useful rules [Minton 1988b]. SOAR implements a criterion requiring that learning occur whenever the system must perform search to resolve an impasse [Laird et al. 1986a].

5.1.3 Which Schemata Should Be Retained?

After a schema is formed, empirical methods can be used to determine whether it is useful in practice [Markovitch and Scott 1988; Minton 1988b; Silver 1988]. Minton's PRODIGY system collects statistics that help to evaluate the "utility" of learned rules. PRODIGY uses a measure of utility that involves (1) the cost of testing a rule for applicability, (2) the frequency with which the rule is applicable, and (3) the savings that results from using the rule. The measure is designed so that rules with positive utility will improve overall performance. Rules are retained in memory only as long as they are estimated to have positive utility.

Mostow and Cohen [1985] have examined this issue in a slightly different context. They have investigated the cost effectiveness of software "caches" that avoid recomputing functions by storing and reusing the results of computations. They list several criteria bearing on the question of which computations should be cached, including the hit rate, the lookup cost, and the cost of the original computation. Cache formation is similar to EBL schema formation inasmuch as both processes implement a "store versus compute trade-off" [Rosenbloom and Newell 1986]. Thus, these criteria may also be relevant for determining when schema formation is warranted and when schemata should be retained.

5.1.4 Schema Optimization

Assuming that a rule is kept in the memory of an EBL system, the cost of using the rule may be diminished through the use of expression simplification techniques. Minton's PRODIGY system performs "compression analysis," which can simplify individual rules and combine several rules into one [Minton 1988b]. Prieditis' PROLEARN program uses partial evaluation to simplify some expressions [Prieditis and Mostow 1987]. SOAR optimizes rules by reordering the conditions [Laird et al. 1986a]. Other optimizations for SOAR are discussed in Tambe and Newell [1988].

5.1.5 Basic Methods for Analyzing Explanations

Numerous techniques have been developed for analyzing explanations to produce chunks or generalized schemata. A summary of these techniques is shown in Table 1. The methods are categorized according to the language used to represent the domain theory from which explanations are built. Methods have been developed for analyzing explanation structures built from STRIPS operators, Horn clauses, and OPS5 operators, among others.[22] Each performs a function conceptually similar to Waldinger's goal regression procedure [Nilsson 1980; Waldinger 1977] or Dijkstra's notion of weakest preconditions

[22] Inasmuch as Horn clauses may be viewed as special cases of STRIPS operators, the methods listed as applying to STRIPS operators should apply to Horn clauses as well, with at most minor modifications.

Table 1. Methods of Analyzing Explanations

Method	Operator language	Reference
EGGS	STRIPS operators	Mooney and Bennett [1986]
STRIPS	STRIPS operators	Fikes et al. [1972]
EBS	STRIPS operators	Minton and Carbonell [1987]
MGR	Horn clauses	Mitchell et al. [1986]
PROLOG-EBG	Horn clauses	Kedar-Cabelli and McCarty [1987]
MRS-EBG	Horn clauses	Hirsh [1987]
PROLEARN	Horn clauses	Prieditis and Mostow [1987]
SOAR	OPS5 operators	Laird et al. [1987]
Unnamed	OPS5 operators	Benjamin [1987]
EGR	Black boxes	Porter and Kibler [1986]

[Dijkstra 1976]. For a comparison of MGR, EGGS, and the STRIPS proof generalizer, see Mooney and Bennett [1986].

Kibler and Porter [1986] have developed a technique called "experimental goal regression" to handle types of operators for which all known analytical methods fail. This empirical technique has the advantage of not requiring access to the internal representation of the operators; however, it has the disadvantage that only approximate weakest preconditions are found. Porter and Kibler [1985] have also presented some criteria that operator description languages must meet in order for analytical goal regression to succeed. Additional research is needed to extend the range of operator languages that can be analyzed and to determine the types of languages that are analyzable in principle.

5.1.6 Enhanced Methods for Analyzing Explanations

A number of techniques have recently been developed to create schemata of greater generality than can be obtained with the methods listed in Table 1. The basic methods generalize explanations into schemata that are provable using the original explanation structure. Thus, the resulting schemata are no more general than the original explanation structure. As observed by DeJong [DeJong and Mooney 1986], it may be necessary in some cases to generalize the explanation structure itself. For example, this need arises in the shift-register example described by Ellman [1985]. The number of cells in the register cannot be generalized by the standard methods because additional cells would require correctness proofs with a greater number of inference steps.

Methods for generalizing the number of inference steps in an explanation are discussed by Shavlik and DeJong [1987a, 1987b], Cohen [1988], Cheng and Carbonell [1986], and Prieditis [1986]. Shavlik's BAGGER system uses a technique that transforms a single rule into a schema representing the effect of repeated applications of the rule. Cohen's ADEPT system analyzes explanations to produce finite-state automata. The automata are used to deterministically guide a theorem prover. Mooney has developed an extension of EGGS that can generalize the order of operators in a macro operator [Mooney 1988]. Given a sequence of STRIPS operators, his method finds the most general partial ordering of the sequence-preserving constraints related to interaction of operator preconditions and effects. Empirical methods for generalizing the explanation structure are discussed in Kedar-Cabelli [1985] and Flann and Dietterich [1986]. By comparing explanation structures taken from multiple examples, commonly used substructures can be identified, extracted, and generalized.

5.1.7 Representation of Domain Theories and Explanations

The results of EBL appear to depend critically on the representation of domain theories and explanations. A number of authors have commented on this relation. Gupta [1988] has identified cases in which the generality of learned rules is influenced by details of the domain theory representation. DeJong has argued that a schema-based representation is important to the success of EBL systems [DeJong and Mooney 1986]. Several authors have observed that EBL generalization can be influenced by the grain size of a domain theory [Braverman and Russell 1988; Roy and Mostow 1988]. Although there appears to be widespread agreement that the representation is critical, very little is known in general about what makes a representation good for the purposes of EBL. Future research might attempt to identify a set of guidelines to be used by people encoding the initial domain theories used in EBL.

5.1.8 Generality and Operationality

Much of the foregoing discussion has assumed that general schemata are to be preferred over specific ones. Although a general schema will have wider applicability than a specific schema, general ones are not always better. In order to apply to new examples, schemata must be instantiated. General schemata are presumably harder to instantiate than specific ones. A more general schema may therefore be deemed less operational owing to the high cost of instantiation. This issue has been discussed by several authors in terms of the "operationality/generality trade-off" [Segre 1987; Shavlik and DeJong 1987c]. Segre presents some preliminary empirical evidence for the existence of this trade-off. Keller argues that the operationality/generality trade-off does not occur in all contexts in which EBL systems can be used [Keller 1988a]. He suggests that generality and operationality should be seen as separate, potentially independent dimensions.

The existence of an operationality/generality trade-off would place two demands on EBL systems. To begin with, EBL systems need the capability of generating schemata at various levels of generality. Methods of controlling schema generality are presented by Segre [1987] and Braverman and Russell [1988]. Given the ability to generate schemata of varying generality, EBL systems also need methods of determining which ones will have the most favorable impact on overall performance. The empirical techniques described above for deciding when to retain schemata may be useful in this context.

5.1.9 Criteria of "Operationality"

Several researchers have offered new methods of defining the term *operationality* for the purpose of explanation-based learning. EBG considers a concept description to be "operational" if it is expressed using predicates drawn from a predefined list of operational predicates; otherwise the description is "nonoperational" (see Section 3.1). DeJong points out that operationality can sometimes depend on the arguments to a predicate, as well as the predicate itself [DeJong and Mooney 1986]. He also argues that operationality can vary with the knowledge contained in the system. DeJong suggests that a goal be considered "operational" whenever the system possesses a schema for solving the goal. As the system acquires more schemata, more goals are considered operational. In contrast to the binary operational/nonoperational distinction used by Mitchell et al. [1986] and DeJong and Mooney [1986], both MetaLEX [Keller 1987a, 1987b] and PRODIGY [Minton 1988a, 1988b] use continuous measures of operationality. Continuous measures seem more appropriate when operationality is intended to capture some notion of computational efficiency.

Keller and Segre have each argued that efficiency alone is not sufficient for defining operationality. Segre suggests a set of criteria would be more appropriate to real-world planning problems [Segre 1988]. His five criteria include efficiency, generality, robustness, recoverability, and obviousness. Keller takes the position that no single set of criteria will be appropriate in all learning situations [Keller 1988b]. He proposes that operationality be defined in relation to the context in which learning occurs. The operationality of a concept description will depend on (1) the performance system using the description and (2) the performance objectives of the system. When the context changes, the definition of operationality changes as well.

Several authors have argued that EBL systems should be given the power to explicitly reason about operationality [DeJong and Mooney 1986; Hirsh 1988a; Keller 1988b]. Mostow's BAR program and Hirsh's ROE program both use special rules to reason about the operationality of expressions [Hirsh 1988a; Mostow 1983a, 1987b]. Keller's MetaLEX and Minton's PRODIGY programs each perform empirical tests of the operationality of concept descriptions [Keller 1987b; Minton 1988b].

5.1.10 *Automatic Formulation of Learning Tasks*

In order to attack the problem of "wandering bottlenecks" [Mitchell 1983], systems must have the ability to automatically formulate their own learning tasks. Keller [1987a] has outlined an approach to this problem in the context of his MetaLEX system (see Section 2.4.2). Taking a description of the performance system and the performance objectives as input, his method would automatically generate the goal concept and operationality criterion used by EBG. Kedar-Cabelli's PurForm system does something similar [Kedar-Cabelli 1987]. PurForm automatically generates a goal concept describing an everyday artifact, when given the "purpose" of the artifact as input. SOAR may also achieve a similar capability to deal with wandering bottlenecks. "Universal subgoaling" is intended to provide a framework within which SOAR can formulate goals to improve any aspect of the system's performance (see Section 2.3). All these approaches require tackling serious knowledge representation problems. The representation must support reasoning about the presence of bottlenecks within the system's performance element, and reasoning about potential methods of alleviating the bottlenecks.

5.1.11 *Interacting with Humans*

A final group of issues involves so-called "Learning Apprentice" programs such as the LEAP system [Mitchell et al. 1985]. LEAP is intended to use EBL methods to learn by watching a human expert in the course of normal problem solving. In order for EBL to be truly useful as a knowledge acquisition tool, it must not make inordinate demands on human experts' time. LEAP requires a human expert to supply the initial domain theory. Building the initial theory may be as difficult for a human as directly building the "expert" theory that the EBL system would produce. Perhaps the cost of building an initial theory could be amortized over time as the theory gets reformulated for a variety of purposes. In any case, only practical experience will determine whether EBL can make economical use of a human expert's time.

Other issues of human–computer interaction must also be addressed. Human experts will inevitably supply examples that are erroneous or suboptimal. An EBL system may be able to use the domain theory to detect and correct such errors; however, it must do so in an unobtrusive manner. DeJong has suggested that an EBL system can use the domain theory to improve suboptimal plans [DeJong and Mooney 1986]. Lebowitz [1986a] has discussed filtering out erroneous examples by combining EBL with empirical learning techniques.

5.2 EBL and Theory Revision

Most EBL methods are based on the assumption that the initial domain theory is adequate to explain all the examples to be processed by the learning system. Although this condition may be met in highly constrained domains, it will not be met in the context of most real-life situations. If the domain theory cannot explain a training example, then the EBG method will fail to operate. Methods must be developed that enable explanation-based learning to proceed in the absence of an adequate domain theory. Of course, this objective is not an end in itself. The real purpose of learning is to improve the system's domain theory. Methods must be developed by which EBL can remedy the deficiencies in an initial domain theory.

An initial domain theory can suffer from a number of distinct deficiencies. A classification of defects is shown in Figure 37. This diagram shows four dimensions along which an initial theory can be evaluated,

Completeness: Does the theory entail *at least* one positive or negative classification for each example in the domain?

Consistency: Does the theory entail *at most* one positive or negative classification for each example?

Correctness: Are all the predictions entailed by the theory actually correct?

Tractability: Can explanations of all examples be constructed without exhausting specified time and space resources?

Figure 37. Classification of theory defects.

including "completeness," "correctness," "consistency," and "tractability." Two similar typologies are outlined by Mitchell et al. [1986] and Rajamoney and DeJong [1987]. These differ from the present typology in failing to distinguish between incomplete and incorrect theories. In addition to classifying imperfect theories, Rajamoney and DeJong enumerate some of the possible causes of incompleteness, inconsistency, and intractability. They also discuss methods of detecting such imperfections.

In some contexts the distinctions between imperfect theory types can become blurred. Consider the case of a nonmonotonic theory, that is, a theory in which the addition of new information can invalidate previously derived explanations. If some rules are missing, the theory may entail conclusions that would be considered incorrect if the missing rules were present. Thus, incompleteness can cause incorrectness. Another ambiguity involves intractable theories. An intractable theory becomes incomplete if explanations exceeding resource limits are forbidden. It becomes incorrect if approximations are used to simplify the theory. Owing to these blurred distinctions, some of the techniques discussed in the following sections may be validly understood as addressing more than one of the imperfect theory types.

5.2.1 Incomplete Domain Theories

A domain theory is considered "incomplete" if it fails to explain some examples from the domain under study. According to this definition, an "incomplete" theory is not necessarily "incorrect." A theory might be perfectly correct as far as it goes. It nevertheless would be incomplete if it fails to say anything about some examples, or some parts of examples. The distinction between incomplete and incorrect theories is important. A correct but incomplete theory presents less difficulty than an incorrect theory. If a theory is merely incomplete, the EBG method may succeed on some examples. If any explanation can be found, it is certain to be correct, and the generalization process can proceed as if the theory were perfectly adequate. In comparison, an incorrect theory might produce an explanation of an example that omits key details, leading to incorrect generalizations.

A number of techniques use "partial explanations" to handle incomplete theories [Berwick 1985; Hall 1988; Pazzani 1988; Roy and Mostow 1988; Sleeman et al. 1987; VanLehn 1987; Wilkins 1988]. When faced with an example that cannot be fully explained, these systems attempt to explain as much as possible. By focusing on gaps in the resulting partial explanations, they identify and conjecture new rules that would make the explanations complete. In most cases, more than one alternative rule can complete an explanation. Methods are therefore needed to guide the generation and evaluation of alternative conjectures. Analytic methods that evaluate alternatives using a separate "confirmation theory" are described by Wilkins [1988] and Hall [1988]. Empirical methods would also appear to be relevant; that is, multiple examples could be used to choose among alternative conjectures [Pazzani 1988]. Partial explanations appear to be most effective when the initial domain theory is nearly complete, or when a teacher carefully selects and orders examples to introduce one new rule at a time [VanLehn 1987]. Inasmuch as these techniques add new rules, but do not revise or retract old ones, they implicitly assume the correctness of the initial, incomplete theory.

Incomplete theories can be handled by approaches that combine both empirical and analytical learning techniques. Given an initial domain theory, one could construct a version space containing only concept descriptions that are consistent with the theory. As long as the initial theory is incomplete, the space will contain more than one concept description. The candidate elimination algorithm [Mitchell 1978] then could be used to process training examples and pare down the set of candidates (see Section 2.2.2). Russell and Grosof have shown how an incomplete theory containing "determinations" can be used to prune an initial, unbiased version space [Russell 1988; Russell and Grosof 1987]. Mahadevan and Tadepalli [1988] have analyzed the information complexity of learning from such incomplete theories represented in the form of determinations.

5.2.2 *Incorrect Domain Theories*

A variety of explanation-based methods have been developed to deal with incorrect domain theories. These methods can be best understood by considering two processes involved in revising an incorrect theory that is found to commit an error. The first step is "blame assignment," that is, identifying the parts of the theory that caused the error. After identifying the faulty portions of the theory, suitable changes to these parts must be found.

Blame assignment is sometimes handled with the technique of tracing dependency links [Bylander and Weintraub 1988; Clancey 1988; Doyle 1979; Pazzani 1988; Smith et al. 1985]. By starting at the erroneous conclusion and tracing backward through the explanation structure, it is possible to identify pieces of domain knowledge that might have caused the error. In most cases, the flawed knowledge cannot be uniquely identified. In order to ameliorate this difficulty, Smith's system uses an enriched representation that can describe "belief types" and "error types," among other things [Smith et al. 1985]. The system propagates error types through the network and can reason about the relation between error types and belief types. As a result, the search for faulty knowledge is more tightly constrained than that in simpler methods of tracing dependencies.

Blame can also be assigned by special domain-dependent rules for identifying faulty knowledge [Carbonell and Gil 1987; Hunter 1988; Rajamoney 1988]. Such rules classify errors and associate possible causes with each error type. Rajamoney's system uses special domain-dependent rules to generate "high-level explanations" of errors in the domain of chemical processes. For example, a high-level explanation might conclude that "some process has erroneous effects" or "some process has erroneous preconditions." Each high-level explanation makes an "abstract hypothesis" about which piece of knowledge is faulty; that is, the fault is characterized in a general way, but not specifically identified.

Blame assignment takes a special form when an incorrect domain theory can be viewed as an "abstraction" or "approximation" of another, more accurate theory, as in Doyle [1986] and Davis [1985]. These systems explicitly represent the approximations or abstractions that were applied to the accurate theory to generate the incorrect one. Blame assignment then reduces to the problem of finding faulty approximations or abstractions. This viewpoint is especially relevant to the problem of diagnosing faults in devices. Diagnosis is sometimes viewed as a problem of revising an ideal model of device behavior by relaxing abstractions underlying the ideal model [Davis 1985].

After faulty parts of an incorrect theory are identified, the theory must be revised. In most cases, more than one revision will be possible. This occurs if blame cannot be uniquely assigned or if faults can be cured in multiple ways. Several people have investigated "experimentation" as a method of resolving such ambiguity [Carbonell and Gil 1987; Rajamoney and DeJong 1988]. Rajamoney's system can propose experiments to discriminate between alternative "abstract hypotheses." It also designs experiments that help select between various remedies after a fault is identified.

5.2.3 *Inconsistent Domain Theories*

An inconsistent domain theory is one that contains statements that lead to logically contradictory predictions. Inconsistency can lead to serious problems if proof by contradiction is used in the system. The problems of inconsistent and incorrect theories are closely related. Both involve "inconsistency" in a broader sense. In one case, the inconsistency is internal to the theory, and in the other case, the inconsistency exists between the theory and observations. For this reason many of the methods for handling incorrect theories should apply to inconsistent theories (e.g., assigning blame by tracing dependencies).

As an example of inconsistency, consider the so-called "promiscuous theories." These theories have the property that they can generate plausible explanations of nearly anything that might be observed. Considering the congressional voting domain as an example, one can imagine explanations of why a liberal senator from Connecticut would vote either "yes" or "no" on a defense-spending bill. As a liberal, he should vote "no." Coming from a state with a large defense industry, he should vote "yes." Lebowitz [1986a] has attacked the problem of inconsistent, promiscuous theories by combining EBL with empirical learning techniques. Riesbeck [1983] has also developed explanation-based methods for dealing with promiscuous theories in the domain of macroeconomics.

5.2.4 *Intractable Domain Theories*

A theory is considered "intractable" if it cannot be used to make predictions or explain observations without consuming inordinate computational resources. For example, consider the game of chess. In order to fully explain a training example from chess, an EBL system would exhaustively search the game tree—a task that is not possible in practice. An intractable theory may be complete and correct in principle. Time and space limitations make it behave as an incomplete theory since many consequences of the theory cannot be inferred in a reasonable length of time.

The intractable theory problem might be solved by finding simplifying assumptions or approximations that make the theory more tractable. By introducing approximations into a correct but intractable theory, one hopes to trade accuracy in return for a gain in efficiency. A difficulty arises when a theory can be approximated in multiple, inconsistent ways. In such cases, empirical methods can be used to determine which of several alternative approximations yields the most accurate predictions.

Quite a variety of approximation types have been considered by investigators taking the approach outlined above. These include (1) making functions or expressions invariant with respect to their arguments, in algebraic domains [Bennett 1987; Ellman 1988; Keller 1987b; Mostow and Fawcett 1987]; (2) assuming the absence of counterplanning by adversaries in planning domains and games [Chien 1987a; Tadepalli 1986]; (3) ignoring constraints limiting possible causal interactions among physical devices [Doyle 1986]; (4) assuming persistence of state variables after state changes [Chien 1987b]; and (5) treating random variables as independent or equiprobable [Ellman 1988].

Two general architectures have been suggested for systems that learn approximations to intractable theories. One involves generating a search space in which each point corresponds to a different approximate theory [Ellman 1988; Keller 1987b; Mostow and Fawcett 1987]. Explanations of training examples are used to guide a search through the approximate theory space. Another approach uses a simple, highly approximate theory until a failure is generated. Analysis of the failure leads to revising or retracting approximations [Chien 1987a, 1988; Doyle 1986; Gupta 1987; Mostow and Bhatnagar 1987; Tadepalli 1986; Zweben and Chase 1988].

5.3 Integrated Learning

A major outstanding problem in machine learning involves the relation between the analytical techniques of explanation-based learning and empirical learning methods. Several reasons for integrating EBL with empirical methods have been discussed in previous sections. Integrated methods were shown to be relevant in the context of "EBL and theory reformulation" for several reasons. These include (1) empirically evaluating the utility of schemata; (2) comparing explanations of multiple examples to find common parts; (3) handling noisy or erroneous examples; and (4) experimental goal regression. In the context of "EBL and theory revision," integrated learning methods have been suggested to remedy each of the types of imperfect theories.

Several investigators have built systems that incorporate both explanation-based and empirical methods in a single architecture. These fall into three main groups: (1) using explanations to process the results of empirical learning [Lebowitz 1986a]; (2) using empirical methods to process the results of an explanation phase [Carpineto 1988; Danyluk 1987; Dietterich and Flann 1988; Flann and Dietterich 1986; Salzberg 1983]; and (3) using integrated combinations of explanation-based and empirical methods [Bergadano and Giordana 1988; Danyluk 1988; Swaminathan 1988]. Hirsh [1988b] describes a framework for characterizing these hybrid systems in terms of a space of possible generalizations. The relative importance of empirical and explanation-based methods is discussed by Lebowitz [1986b] and Pazzani et al. [1986].

In order to illuminate such hybrid systems, it helps to consider the ways in which empirical and explanation-based methods can enhance each other. First consider how empirical methods can improve explanation-based learning. When empirical methods are used *before* an explanation phase, the process of building an explanation can become computationally simpler. This occurs in Lebowitz's UNIMEM system [Lebowitz 1986a], which uses empirical methods to filter out erroneous and irrelevant features that would slow down the explanation process. When empirical methods are used *after* an explanation phase, they can filter out erroneous explanations that are inconsistent with the empirical data [Dietterich and Flann 1988].

Now consider how explanations can enhance empirical learning methods. When the explanation phase occurs *before* empirical learning, the explanations can improve the representation of training examples. Explanations can be used to derive features that are only implicit in the training data [Buchanan and Mitchell 1978; Carpineto 1988; Danyluk 1987; Flann and Dietterich 1986; Mitchell 1983]. They can also be used to remove irrelevant features [Danyluk 1987; Salzberg 1983]. When the explanation phase occurs *after* an empirical phase, it can act as a filter on the results of empirical learning. By discarding empirical generalizations that cannot be explained, the results will be less influenced by coincidental correlations in the data.

No one has yet formulated a principled method of combining explanation-based and empirical learning methods. Such principles might well result from analyzing the relation between EBL and the inductive bias used by empirical techniques. As suggested above, EBL may be equivalent to a "bias toward explainability" represented in terms of a declarative domain model (see Section 1). This relation might be clarified by trying to express traditional types of inductive bias in a declarative manner. Some initial efforts in this direction have been made by Russell and Grosof [Russell 1988; Russell and Grosof 1987] and by Dietterich [1986]. If biased generalization languages and algorithms are expressed in a declarative representation, they might be equated with the initial domain theory used in explanation-based learning. The distinction between explanation-based and empirical methods would be reduced to an instance of the declarative/procedural controversy.

6. SUMMARY

This paper has provided an overview of the field of explanation-based learning. The EBL field was placed in the context of other knowledge-intensive approaches to machine learning. EBL was described as a merging of four trends in machine learning research, including generalization, chunking, operationalization, and analogy. Examples of EBL programs from each of these areas were discussed. In this paper an attempt to formally define EBL methods and the problems they can handle was also described. The formalization raises fundamental questions about the types of learning that EBL can and cannot perform. Directions for future research were also discussed. Three main areas for future work are "EBL and theory reformulation," "EBL and theory revision," and "integrated learning."

7. GLOSSARY OF SELECTED TERMS

Analytic learning: Any learning method that relies mainly on processing preexisting background knowledge and requires few, if any, externally provided examples (contrasts with empirical learning) [Mitchell 1982b] (Introduction).

Bias: Any criteria used by a concept learning program to choose among alternative generalizations that are each consistent with the observed training instances [Mitchell 1980] (Section 1.1).

CBP: See Constraint back-propagation.

Chunking: In the context of explanation-based learning, chunking is a process of compiling a linear or tree-structured sequence of operators into a single operator. The single operator has the same effect as the entire original sequence [Rosenbloom and Newell 1986] (Sections 1.2 and 2.3).

Constraint back-propagation (CBP): A procedure used in LEX-II for analyzing goals and operator sequences. Given an operator OP and a goal pattern P, CBP finds a new pattern P′ such that the state OP(S) matches P if and only if S matches P′ [Utgoff 1986] (Section 2.2.2).

Contextual knowledge: Knowledge of the context in which learning takes place. Contextual knowledge has several components [Keller 1987a], including (1) a description of the performance element to be improved by learning and (2) a specification of performance objectives, among other things (Section 2.4.2).

EBA: See Explanation-based analogy.

EBG: See Explanation-based generalization.

EBL: See Explanation-based learning.

EGGS: An algorithm used in GENESIS for generalizing explanations [Mooney and Bennett 1986]. Given an explanation structure ES as input, EGGS finds the most general instantiation of ES that represents a valid explanation (Section 2.2.1).

Empirical learning: Any learning technique that relies mainly on examining multiple, externally provided training examples and requires little or no background knowledge of the domain under study (contrasts with Analytic learning) [Langley 1986; Mitchell 1982b] (Introduction and Section 5.3).

Explanation-based analogy (EBA): A method of analogical reasoning that involves transferring explanations, derivations, or networks of causal relations from analogs to target examples (Section 2.5).

Explanation-based generalization (EBG): A formalism that attempts to capture the elements of most explanation-based learning programs. EBG takes as input a domain theory, a goal concept, an operationality criterion, and a training example. It finds an operational concept description that includes the example and is a sufficient condition for the goal concept [Mitchell et al. 1986] (Section 3.1).

Explanation-based learning (EBL): A type of analytic learning, the definition of which constitutes the subject of this paper (Introduction). EBL is intended to include explanation-based generalization and explanation-based analogy, as well as certain types of chunking and operationalization (it may also include explanation-based concept specialization [DeJong and Mooney 1986]).

Explanation structure: An "overgeneralized" explanation that results from replacing each instantiated rule in an explanation with the associated general rule (using unique variables for distinct rule applications) [Mitchell et al. 1986; Mooney and Bennett 1986] (Sections 2.2.1 and 3.1).

Generalization: The word *generalization* can refer to either a concept or a process of concept formation. In the first sense of the word, a concept is a "generalization" of an example if it includes the example. In the second sense of the word, "generalization" is a process that takes one or more training examples as input and produces a concept that includes all the positive examples and excludes all the negative examples. A discussion of the term *generalization* is found in Langley [1986] (Introduction and Section 2.2).

Goal regression: A procedure for analyzing sequences of STRIPS operators. Given an operator sequence S and a goal G, a goal regression finds a condition C such that, for any state X, APPLY(S, X) satisfies G if and only if X satisfies C [Nilsson 1980; Waldinger 1977] (Sections 2.2.2, 3.1, and 5.1).

Justified analogy: A logically sound procedure for reasoning by analogy. Given some initial background knowledge B, an analog example X, and a target example Y, find a feature F such that F(X) is true, and infer that F(Y) is true. The conclusion F(Y) must be a logical consequence of F(X) and the background knowledge B [Davies and Russell 1987] (Sections 1.2 and 2.5).

Justified generalization: A logically sound procedure for generalizing from examples. Given some initial background knowledge B and a set of training examples T, justified generalization finds a concept C that includes all the positive examples and excludes all the negative examples. The learned concept C must be a logical consequence of the background knowledge B and the training example set T [Russell 1986] (Sections 1.2 and 2.2).

Knowledge-level learning: A system is said to perform knowledge-level learning when there is a change over time of the deductive closure of its knowledge. The deductive closure of a set of axioms is defined to include the axioms themselves, plus all facts derivable from the axioms using an arbitrary number of inference steps [Dietterich 1986] (Section 4).

MGR: See Modified goal regression.

Modified goal regression (MGR): A procedure used in explanation-based generalization (EBG) to analyze proof trees [Mitchell et al. 1986]. Given a goal concept literal G and an explanation (tree) structure T, MGR finds a set of generalized antecedents A. Any instance of A can be proven to satisfy the goal concept G using the explanation (tree) structure T (Section 3.1).

Operationalization: A process of translating a nonoperational expression into an operational one. The initial nonoperational expression may be a set of instructions (as in operationalizing advice [Mostow 1983a, 1983c]) or a concept (as in concept operationalization [Keller 1983]). Concepts and instructions are considered to be operational with respect to an agent if they are expressed in terms of actions and data available to the agent [Mostow 1983a] (Sections 1.2 and 2.4).

Similarity-based learning: A synonym for Empirical learning (see Empirical learning). The term *similarity-based learning* is defined by Lebowitz [1986a]. Issues related to the use of this term are discussed by Langley [1986] (Introduction and Section 5.3).

Weak method: A problem-solving technique that can be used when specific domain knowledge is not available. Examples include means–ends analysis and hill climbing, among others [Newell 1969] (Section 2.3.1).

ACKNOWLEDGMENTS

Many thanks are due to Michael Lebowitz for numerous useful discussions about the material in this paper. This research was supported in part by the Defense Advanced Research Projects Agency under Contract N00039-84-C-0165.

REFERENCES

AHN, W., MOONEY, R. J., BREWER, W. F., AND DEJONG, G. F. 1987. Schema acquisition from one example: Psychological evidence for explanation-based learning. Tech. Rep. UILU-ENG-87-2231, Coordinated Science Laboratory, Univ. of Illinois, Urbana–Champaign, Ill.

AMAREL, S. 1968. On representations of problems of reasoning about actions. In *Machine Intelligence 3*, D. Michie, Ed. American Elsevier, New York, pp. 131–171.

ANDERSON, J. R. 1983a. *The Architecture of Cognition.* Harvard Univ. Press, Cambridge, Mass.

ANDERSON, J. R. 1983b. Acquisition of proof skills in geometry. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 191–219.

ANDERSON, J. R. 1986. Knowledge compilation: The general learning mechanism. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 289–310.

ANGLUIN, D., AND SMITH, C. H. 1983. Inductive inference: Theory and methods. *ACM Comput. Surv. 15*, 3 (Sept.), 237–269.

BALZER, R. GOLDMAN, N., AND WILE, D. 1976. On the transformational implementation approach to programming. In *Proceedings of the 2nd International Conference on Software Engineering.* IEEE, New York, pp. 337–343.

BANERJI, R. B., AND ERNST, G. W. 1972. Strategy construction using homomorphisms between games. *Artif. Intell. 3*, 223–249.

BENJAMIN, D. P. 1987. Learning strategies by reasoning about rules. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 256–259.

BENNETT, S. W. 1987. Approximation in mathematical domains. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 239–241.

BERGADANO, F., AND GIORDANA, A. 1988. A knowledge intensive approach to concept induction. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 305–317.

BERWICK, R. C. 1985. *The Acquisition of Syntactic Knowledge.* MIT Press, Cambridge, Mass.

BRAVERMAN, M. S., AND RUSSELL, S. J. 1988. Boundaries of operationality. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 221–234.

BUCHANAN, B. G., AND MITCHELL, T. M. 1978. Model-directed learning of production rules. In *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth, Eds. Academic Press, Orlando, Fla., pp. 297–312.

BYLANDER, T., AND WEINTRAUB, M. A. 1988. A corrective learning procedure using different explanatory types. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 27–30.

CARBONELL, J. G. 1983a. Derivational analogy and its role in problem solving. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 64–69.

CARBONELL, J. G. 1983b. Learning by analogy: Formulating and generalizing plans from past experience. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 137–161.

CARBONELL, J. G. 1986. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 371–392.

CARBONELL, J. G., AND GIL, Y. 1987. Learning by experimentation. In *Proceedings of the 4th International Workshop on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 256–266.

CARPINETO, C. 1988. An approach based on integrated learning to generating stories from stories. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 298–304.

CHENG, P. W., AND CARBONELL, J. G. 1986. The Fermi system: Inducing iterative macro-operators from experience. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 490–495.

CHIEN, S. A. 1987a. Extending explanation-based learning: Failure-driven schema refinement. Tech. Rep. UILU-ENG-87-2203, Coordinated Science Laboratory, Univ. of Illinois, Urbana–Champaign, Ill.

CHIEN, S. A. 1987b. Simplifications in temporal persistence: An approach to the intractable domain theory problem in explanation-based learning. Tech. Rep. UILU-ENG-87-2255, Coordinated Science Laboratory, Univ. of Illinois, Urbana–Champaign, Ill.

CHIEN, S. A. 1988. A framework for explanation-based refinement. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 137–141.

CLANCEY, W. J. 1988. Detecting and coping with failure. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 22–26.

COHEN, P. R., AND FEIGENBAUM, E. A. (Eds.) 1982. *The Handbook of Artificial Intelligence, Volume 3.* William Kaufmann, Inc., Los Altos, Calif., pp. 323–511.

COHEN, W. W. 1988. Generalizing number and learning from multiple examples in explanation-based learning. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 256–269.

CULLINGFORD, R. 1978. Script application: Computer understanding of newspaper stories. Tech. Rep. 116, Dept. of Computer Science, Yale Univ., New Haven, Conn.

DANYLUK, A. P. 1987. The use of explanations for similarity-based learning. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 274–276.

DANYLUK, A. P. 1988. Integrated learning is a two way street. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 36–40.

DAVIES, T. R., AND RUSSELL, S. J. 1987. A logical approach to reasoning by analogy. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 264–270.

DAVIS, R. 1985. Diagnostic reasoning based on structure and behavior. In *Qualitative Reasoning about Physical Systems*, D. G. Bobrow, Ed. MIT Press, Cambridge, Mass., pp. 347–410. Also found in *Artif. Intell. 24.*

DAWSON, C., AND SIKLOSSY, L. 1977. The role of preprocessing in problem solving systems. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 465–471.

DEJONG, G. F. 1981. Generalizations based on explanations. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 67–69.

DEJONG, G. F. 1986. An approach to learning from observation. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 571–590.

DEJONG, G. F., AND MOONEY, R. 1986. Explanation-based learning: An alternative view. *Mach. Learn. 1*, 2, 145–176.

DIETTERICH, T. G. 1986. Learning at the knowledge level. *Mach. Learn. 1*, 3, 287–315.

DIETTERICH, T. G., AND FLANN, N. S. 1988. An inductive approach to the imperfect theory problem. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 42–46.

DIETTERICH, T. G., AND MICHALSKI, R. S. 1981. Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods. *Artif. Intell. 16*, 257–294.

DIJKSTRA, E. W. 1976. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J.

DOYLE, J. 1979. A truth maintenance system. *Artif. Intell. 12*, 231–272.

DOYLE, R. J. 1986. Constructing and refining causal explanations for an inconsistent domain theory. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 538–544.

ELLMAN, T. 1985. Generalizing logic circuit designs by analyzing proofs of correctness. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 643–646.

ELLMAN, T. 1988. Approximate theory formation: An explanation-based approach. In *Proceedings of the 7th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 571–574.

FIKES, R. E., HART, P. E., AND NILSSON, N. J. 1972. Learning and executing generalized robot plans. *Artif. Intell. 3*, 251–288.

FLANN, N. S., AND DIETTERICH, T. G. 1986. Selecting appropriate representations for learning from examples. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif.

FORGY, C. L. 1981. OPS5 user's manual. Tech. Rep. CMU-CS-81-135, Dept. of Computer Science. Carnegie-Mellon Univ., Pittsburgh, Pa.

GENTNER, D. 1983. Structure mapping: A theoretical framework for analogy. *Cognitive Sci. 7*, 155–170.

GIBSON, W. B. 1974. *Hoyle's Modern Encyclopedia of Card Games.* Doubleday, Garden City, N.Y.

GUPTA, A. 1987. Explanation-based failure recovery. In *Proceedings of the 6th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 606–610.

GUPTA, A. 1988. Significance of the explanation language in EBL. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 73–77.

HALL, R. J. 1988. Learning by failing to explain: Using partial explanations to learn in incomplete or intractable domains. *Mach. Learn. 3*, 1, 45–77.

HAMMOND, K. J. 1986. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 556–560.

HAMMOND, K. J. 1987. Learning and reusing explanations. In *Proceedings of the 4th International Workshop on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 141–147.

HAYES-ROTH, F., AND MOSTOW, D. J. 1981. Machine transformation of advice into a heuristic search procedure. In *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Erlbaum, Hillsdale, N.J., pp. 231–253.

HILL, W. L. 1987. Machine learning for software reuse. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 338–344.

HIRSH, H. 1987. Explanation-based generalization in a logic programming environment. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 221–227.

HIRSH, H. 1988a. Reasoning about operationality for explanation-based learning. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 214–220.

HIRSH, H. 1988b. Empirical techniques for repairing imperfect theories. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 57–61.

HUHNS, M. N., AND ACOSTA, R. D. 1987. Argo: An analogical reasoning system for solving design problems. Tech. Rep. AI/CAD-092-87, Microelectronics and Computer Technology Corporation, Austin, Tex.

HUNTER, L. 1988. Explanation-based discovery. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 2-7.

IBA, G. A. 1985. Learning by discovering macros in puzzle solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 640-642.

KEDAR-CABELLI, S. T. 1985. Purpose-directed analogy. In *Proceedings of the 7th Annual Conference of the Cognitive Science Society.* Lawrence Carlbaum, Hillsdale, N.J.

KEDAR-CABELLI, S. T. 1987. Formulating concepts according to purpose. In *Proceedings of the 6th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 477-481.

KEDAR-CABELLI, S. T., AND MCCARTY, L. T. 1987. Explanation-based generalization as resolution theorem proving. In *Proceedings of the 4th International Workshop on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 383-389.

KELLER, R. M. 1983. Learning by re-expressing concepts for efficient recognition. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 182-186.

KELLER, R. M. 1987a. The role of explicit contextual knowledge in learning concepts to improve performance. Ph.D. thesis and Tech. Rep. ML-TR-7, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

KELLER, R. M. 1987b. Concept learning in context. In *Proceedings of the 4th International Workshop on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 91-102.

KELLER, R. M. 1988a. Operationality and generality in explanation-based learning: Separate dimensions or opposite endpoints? In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 153-157.

KELLER, R. M. 1988b. Defining operationality for explanation-based learning. *Artif. Intell. 35,* 227-241.

KORF, R. E. 1985. Macro operators: A weak method for learning. *Artif. Intell. 26,* 35-77.

LAIRD, J. E. 1984. Universal subgoaling. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

LAIRD, J. E., AND NEWELL, A. 1983a. A universal weak method: Summary of results. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 771-773.

LAIRD, J. E., AND NEWELL, A. 1983b. A universal weak method. Tech. Rep. CMU-CS-83-141, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

LAIRD, J. E., NEWELL, A., AND ROSENBLOOM, P. S. 1987. SOAR: Architecture for general intelligence. *Artif. Intell. 33,* 1-64.

LAIRD, J. E., ROSENBLOOM, P. S., AND NEWELL, A. 1984. Towards chunking as a general learning mechanism. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 188-192.

LAIRD, J. E., ROSENBLOOM, P. S., AND NEWELL, A. 1986a. Chunking in SOAR: The anatomy of a general learning mechanism. *Mach. Learn. 1,* 1, 11-46.

LAIRD, J. E., ROSENBLOOM, P. S., AND NEWELL, A. 1986b. Over-generalization during knowledge compilation in SOAR. In *Proceedings of the Workshop on Knowledge Compilation.* Department of Computer Science, Oregon State University, Otter Crest, Or., Sept 24-26, pp. 46-57.

LANGLEY, P. 1986. The terminology of machine learning. *Mach. Learn. 1,* 2, 141-144.

LEBOWITZ, M. 1983. Generalization from natural language text. *Cognitive Sci. 7,* 1, 1-40.

LEBOWITZ, M. 1986a. Integrated learning: Controlling explanation. *Cognitive Sci. 10,* 2, 219-240.

LEBOWITZ, M. 1986b. Not the path to perdition: The utility of similarity-based learning. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 533-537.

LENAT, D. B. 1982. AM: Discovery in mathematics as heuristic search. In *Knowledge-Based Systems in Artificial Intelligence,* R. Davis and D. B. Lenat, Eds. McGraw-Hill, New York, pp. 1-225.

LENAT, D. B., PRAKASH, M., AND SHEPHERD, M. 1986. CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Mag. 6,* 4, 65-85.

LEWIS, C. H. 1978. Production system models of practice effects. Ph.D. thesis, Computer Science Dept., Univ. of Michigan, Ann Arbor, Mich.

MAHADEVAN, S. 1985. Verification-based learning: A generalization strategy for inferring problem-decomposition methods. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 616-623.

MAHADEVAN, S., AND TADEPALLI, P. 1988. On the tractability of learning from incomplete theories. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 235-241.

MANO, M. M. 1976. *Computer System Architecture.* Prentice-Hall, Englewood Cliffs, N.J.

MARKOVITCH, S., AND SCOTT, P. D. 1988. The role of forgetting in learning. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 459-465.

MCCARTHY, J. 1968. Programs with common sense. In *Semantic Information Processing,* M. Minsky, Ed. MIT Press, Cambridge, Mass., pp. 403-418.

MICHALSKI, R. S. 1980. Pattern recognition as rule-guided inductive inference. *IEEE Trans. Pattern Anal. Mach. Intell. 2*, 4, 349–361.

MICHALSKI, R. S. 1983. A theory and methodology of inductive learning. *Artif. Intell. 20*, 111–161.

MICHALSKI, R. S., CARBONELL, J. G., AND MITCHELL, T. M., Eds. 1983. *Machine Learning: An Artificial Intelligence Approach.* Morgan Kaufmann, Los Altos, Calif.

MILLER, G. A. 1956. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev. 63*, 81–97.

MINTON, S. 1984. Constraint-based generalization: Learning game playing plans from single examples. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 251–254.

MINTON, S. 1985. Selectively generalizing plans for problem-solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 596–599.

MINTON, S. 1988a. Learning effective search control knowledge: An explanation-based approach. Ph.D. thesis and Tech. Rep. CMU-CS-88-133, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

MINTON, S. 1988b. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the 7th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 564–569.

MINTON, S., AND CARBONELL, J. G. 1987. Strategies for learning search control rules: An explanation-based approach. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 228–235.

MITCHELL, T. M. 1978. Version spaces: An approach to concept learning. Tech. Rep. HPP-79-2, Computer Science Dept., Stanford Univ., Palo Alto, Calif.

MITCHELL, T. M. 1980. The need for biases in learning generalizations. Tech. Rep. CBM-TR-117, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

MITCHELL, T. M. 1982a. Generalization as search. *Artif. Intell. 18*, 203–226.

MITCHELL, T. M. 1982b. Toward combining empirical and analytical methods for inferring heuristics. Tech. Rep. LCSR-TR-27, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

MITCHELL, T. M. 1983. Learning and problem solving. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 1139–1151.

MITCHELL, T. M., KELLER, R. M., AND KEDAR-CABELLI, S. T. 1986. Explanation-based learning: A unifying view. *Mach. Learn. 1*, 1, 47–80.

MITCHELL, T. M., MAHADEVAN, S., AND STEINBERG, L. I. 1985. LEAP: A learning apprentice system for VLSI design. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 573–580.

MITCHELL, T. M., UTGOFF, P. E., AND BANERJI, R. 1983a. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 163-190.

MITCHELL, T. M., UTGOFF, P. E., NUDEL, B., AND BANERJI, R. 1981. Learning problem solving heuristics through practice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 127–134.

MITCHELL, T. M., et al. 1983b. An intelligent aid for circuit redesign. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 274–278.

MOONEY, R. J. 1985. Generalizing explanations of narratives into schemata. M.S. thesis and Tech. Rep. T-159, Coordinated Science Laboratory, Univ. of Illinois, Urbana–Champaign, Ill.

MOONEY, R. J. 1988. Generalizing the order of operators in macro-operators. In *Proceedings of the 5th International Conference on Machine Learning.* Morgan Kaufmann, Los Altos, Calif., pp. 270–283.

MOONEY, R. J., AND BENNETT, S. W. 1986. A domain independent explanation-based generalizer. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 551–555.

MOONEY, R., AND DEJONG, G. F. 1985. Learning schemata for natural language processing. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 681–687.

MOSTOW, D. J. 1981. Mechanical transformation of task heuristics into operational procedures. Ph.D. thesis and Tech. Rep. CMU-CS-81-113, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

MOSTOW, D. J. 1983. Machine transformation of advice into a heuristic search procedure. In *Machine Learning: An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., 367–403.

MOSTOW, J. 1983a. Operationalizing advice: A problem-solving model. In *Proceedings of the International Workshop on Machine Learning.* Dept. of Computer Science, Univ. Illinois, Urbana, Ill., pp. 110–116.

MOSTOW, J. 1983b. A problem solver for making advice operational. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 279–283.

MOSTOW, J. 1986. Why are design derivations hard to replay? In *Machine Learning: A Guide to Cur-*

rent Research, T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, Eds. Kluwer, Norwell, Mass.

MOSTOW, J. 1987a. Design by derivational analogy: Issues in the automated replay of design plans. Tech. Rep. ML-TR-22, Computer Science Dept., Rutgers Univ., New Brunswick, N.J.

MOSTOW, J. 1987b. Searching for operational concept descriptions in BAR, MetaLEX and EBG. In *Proceedings of the 4th International Workshop on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 376–382.

MOSTOW, J., AND BARLEY, M. 1987. Automated reuse of design plans. Tech. Rep. ML-TR-14, Computer Science Dept., Rutgers Univ., New Brunswick, N.J.

MOSTOW, J., AND BHATNAGAR, N. 1987. Failsafe—A floor planner that uses EBG to learn from its failures. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 249–255.

MOSTOW, J., AND COHEN, D. 1985. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 165–172.

MOSTOW, J., AND FAWCETT, T. 1987. Approximating intractable theories: A problem space model. Tech. Rep. ML-TR-16, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

NATARAJAN, B. K., AND TADEPALLI, P. 1988. Two new frameworks for learning. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 402–415.

NEVES, D. M., AND ANDERSON, J. R. 1981. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Erlbaum, Hillsdale, N.J.

NEWELL, A. 1969. Heuristic programming: Ill-structured problems. In *Progress in Operations Research, III*, J. Aronofsky, Ed. Wiley, New York.

NEWELL, A. 1980. Reasoning, problem solving, and decision processes: The problem space as a fundamental category. In *Attention and Performance VIII*, R. Nickerson, Ed. Erlbaum, Hillsdale, N.J.

NILSSON, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Publ., Palo Alto, Calif.

O'RORKE, P. 1984. Generalization for explanation-based schema acquisition. In *Proceedings of the National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 260–263.

O'RORKE, P. 1986. Recent progress on the mathematician's apprentice project. In *Machine Learning: A Guide to Current Research*, T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, Eds. Kluwer, Nowell, Mass.

O'RORKE, P. 1987. LT revisited: Experimental results of applying explanation-based learning to the logic of the Principia Mathematica. In *Proceedings of the 4th International Workshop on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 148–159.

PARTSCH, H., AND STEINBRÜGGEN, R. 1983. Program transformation systems. *ACM Comput. Surv. 15*, 3 (Sept.).

PAZZANI, M. J. 1988. Integrated learning with incorrect and incomplete theories. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 291–297.

PAZZANI, M. J., DYER, M., AND FLOWERS, M. 1986. The role of prior causal theories in generalization. In *Proceedings of the 5th National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 545–550.

PORTER, B. W., AND KIBLER, D. F. 1985. A comparison of analytic and experimental goal regression for machine learning. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 555–559.

PORTER, B. W., AND KIBLER, D. F. 1986. Experimental goal regression: A method for learning problem solving heuristics. *Mach. Learn. 1*, 3, 249–285.

PRIEDITIS, A. E. 1986. Discovery of algorithms from weak methods. In *Proceedings of the International Meeting on Advances in Learning* (July), Les Arc, France.

PRIEDITIS, A. E. 1988a. Environment-guided program transformation. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. AAAI, Menlo Park, Calif., pp. 201–209.

PRIEDITIS, A. E., Ed. 1988b. *Analogica*. Morgan Kaufmann, Los Altos, Calif.

PRIEDITIS, A. E., AND MOSTOW, J. 1987. PROLEARN: Towards a Prolog interpreter that learns. In *Proceedings of the 6th National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 494–498.

RAJAMONEY, S. A. 1988. Experimentation-based theory revision. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. AAAI, Menlo Park, Calif., pp. 7–110.

RAJAMONEY, S. A., AND DEJONG, G. F. 1987. The classification, detection and handling of imperfect theory problems. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 205–207.

RAJAMONEY, S. A., AND DEJONG, G. F. 1988. Active explanation reduction: An approach to the multiple explanations problem. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 242–255.

RIESBECK, C. K. 1983. Knowledge reorganization and reasoning style. Tech. Rep. YALEU/DCS/RR#270, Dept. of Computer Science, Yale Univ., New Haven, Conn.

ROSENBLOOM, P. S. 1983. The chunking of goal hierarchies: A model of practice and stimulus–response compatibility. Ph.D. thesis and Tech. Rep. CMU-TR-83-148, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa.

ROSENBLOOM, P. S., AND LAIRD, J. E. 1986. Mapping explanation-based generalization onto SOAR. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 561–567.

ROSENBLOOM, P. S., AND NEWELL, A. 1986. The chunking of goal hierarchies: A generalized model of practice. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 247–288.

ROY, S., AND MOSTOW, J. 1988. Parsing to learn fine grained rules. In *Proceedings of the 7th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 547–551.

RUSSELL, S. J. 1986. Preliminary steps toward the automation of induction. In *Proceedings of the 5th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 477–484.

RUSSELL, S. J. 1988. Tree-structured bias. In *Proceedings of the 7th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 641–645.

RUSSELL, S. J., AND GROSOF, B. N. 1987. A declarative approach to bias in concept learning. In *Proceedings of the 6th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 505–510.

SALZBERG, S. 1983. Generating hypotheses to explain prediction failures. In *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 352–355.

SCHANK, R. C. 1982. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge Univ. Press, Cambridge, U.K.

SCHANK, R. C. 1987. *Explanation Patterns.* Cambridge Univ. Press, Cambridge, U.K.

SCHANK, R. C., AND ABELSON, R. P. 1977. *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures.* Erlbaum, Hillsdale, N.J.

SEGRE, A. M. 1986. Explanation-based manipulator learning. In *Machine Learning: A Guide to Current Research,* T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, Eds. Kluwer, Norwell, Mass.

SEGRE, A. M. 1987. On the operationality/generality trade-off in explanation-based learning. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 242–248.

SEGRE, A. M. 1988. Operationality and real world plans. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 158–163.

SEGRE, A. M., AND DEJONG, G. F. 1985. Explanation-based manipulator learning: Acquisition of planning ability through observation. In *Proceedings of the IEEE International Conference on Robotics and Automation.* IEEE, New York, pp. 555–560.

SHAVLIK, J. W. 1985. Learning about momentum conservation. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 667–669.

SHAVLIK, J. W. 1986. Learning classical physics. In *Machine Learning: A Guide to Current Research,* T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, Eds. Kluwer, Norwell, Mass.

SHAVLIK, J. W., AND DEJONG, G. F. 1987a. An explanation-based approach to generalizing number. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 236–238.

SHAVLIK, J. W., AND DEJONG, G. F. 1987b. BAGGER: An EBL system that extends and generalizes explanations. In *Proceedings of the 6th National Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., pp. 516–520.

SHAVLIK, J. W., AND DEJONG, G. F. 1987c. Acquiring special case schemata in explanation-based learning. In *Proceedings of the 9th Annual Conference of the Cognitive Science Society* (July), Seattle, Wash., pp. 851–860.

SILVER, B. 1986a. Precondition analysis: Learning control information. In *Machine Learning: An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 647–670.

SILVER, B. 1986b. Learning control information. In *Machine Learning: A Guide to Current Research,* T. M. Mitchell, J. G. Carbonell, and R. S. Michalski, Eds. Kluwer, Norwell, Mass.

SILVER, B. 1988. A hybrid approach in an imperfect domain. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning.* AAAI, Menlo Park, Calif., pp. 52–56.

SLEEMAN, D., HIRSH, H., AND KIM, I. 1987. Expanding an incomplete domain theory: Two case studies. Tech. Rep. AUCS-TR8704, Univ. of Aberdeen, Aberdeen, Scotland.

SMITH, R. G., WINSTON, H. A., MITCHELL, T. M., AND BUCHANAN, B. G. 1985. Representation and use of explicit justifications for knowledge base refinement. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, Los Altos, Calif., 673–680.

SOLOWAY, E. M. 1978. Learning = Interpretation + Generalization: A case study in knowledge-directed learning. Ph.D. thesis and Tech. Rep. COINS-TR-78-13, Computer and Information Science Dept., Univ. of Massachusetts, Amherst, Mass.

STALLMAN, R. M., AND SUSSMAN, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell. 9*, 135–196.

STEINBERG, L. I., AND MITCHELL, T. M. 1985. The redesign system: A knowledge-based approach to VLSI CAD. *IEEE Des. Test Comput. 2*, 1, 45–54.

STEPP, R. E., AND MICHALSKI, R. S. 1986. Conceptual clustering: Inventing goal oriented classifications of structured objects. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 471–498.

SUBRAMANIAN, D., AND SMITH, D. 1988. Knowledge level learning: An alternative view. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. AAAI, Menlo Park, Calif., pp. 196–200.

SUSSMAN, G. J. 1975. *A Computer Model of Skill Acquisition*. American Elsevier, New York.

SWAMINATHAN, K. 1988. Integrated learning with an incomplete and intractable domain theory: The problem of epidemiological diagnosis. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. AAAI, Menlo Park, Calif.

SWARTOUT, W. R. 1983. XPLAIN: A system for creating and explaining expert consulting programs. *Artif. Intell.* 21, 285–325.

TADEPALLI, P. V. 1986. Learning approximate plans in games. Tech. Rep. ML-TR-8, Computer Science Dept., Rutgers Univ., New Brunswick, N.J.

TAMBE, M., AND NEWELL, A. 1988. Some chunks are expensive. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, Los Altos, Calif., pp. 451–458.

UTGOFF, P. E. 1986. Shift of bias for inductive concept learning. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, Los Altos, Calif., pp. 107–148.

UTGOFF, P. E., AND MITCHELL, T. M. 1982. Acquisition of appropriate bias for inductive concept learning. In *Proceedings of the National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 414–417.

Van Harmelen, F., and Bundy, A. 1988. Explanation-based generalization—partial evaluation. *Artif. Intell.* 36, 401–412.

VANLEHN, K. 1987. Learning one subprocedure per lesson. *Artif. Intell. 31*, 1–40.

VERE, S. A. 1977. Induction of relational productions in the presence of background information. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 349–355.

WALDINGER, R. 1977. Achieving several goals simultaneously. In *Machine Intelligence 8*, E. Elcock and D. Michie, Eds. Ellis Horwood, London.

WILENSKY, R. 1978. Understanding goal-based stories. Tech. Rep. 140, Dept. of Computer Science, Yale Univ., New Haven, Conn.

WILKINS, D. C. 1988. Knowledge base refinement using apprenticeship learning techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 646–651.

WINSTON, P. H. 1972. Learning structural descriptions from examples. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York.

WINSTON, P. H. 1982. Learning new principles from precedents and exercises. *Artif. Intell. 19*, 321–350.

WINSTON, P. H., BINFORD, T. O., KATZ, B., AND LOWRY, M. 1983. Learning physical descriptions from functional definitions, examples, and precedents. In *Proceedings of the National Conference on Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., pp. 433–439.

ZWEBEN, M., AND CHASE, M. P. 1988. Improving operationality with approximate heuristics. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*. AAAI, Menlo Park, Calif., pp. 100–106.

Received June 1987; final revision accepted October 1988.