Explanation-Based Learning: A Problem Solving Perspective

Steven Minton*, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni and Yolanda Gil

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

ABSTRACT

This article outlines explanation-based learning (EBL) and its role in improving problem solving performance through experience. Unlike inductive systems, which learn by abstracting common properties from multiple examples, EBL systems explain why a particular example is an instance of a concept. The explanations are then converted into operational recognition rules. In essence, the EBL approach is analytical and knowledge-intensive, whereas inductive methods are empirical and knowledge-poor. This article focuses on extensions of the basic EBL method and their integration with the PRODIGY problem solving system. PRODIGY's EBL method is specifically designed to acquire search control rules that are effective in reducing total search time for complex task domains. Domain-specific search control rules are learned from successful problem solving decisions, costly failures, and unforeseen goal interactions. The ability to specify multiple learning strategies in a declarative manner enables EBL to serve as a general technique for performance improvement. PRODIGY's EBL method is analyzed, illustrated with several examples and performance results, and compared with other methods for integrating EBL and problem solving.

1. Introduction

Learning from examples has long been a focus of machine learning research. Early work in this area focused primarily on inductive methods, which compare many positive and negative instances of the desired concept in order to extract a general concept description [43]. However, in recent years many researchers have focused their attention on analytical methods, which view instances of concepts as more than independent collections of features; they consider each example in the context of background knowledge. Some of the analytical methods most intensively investigated fall under the label of explanation-based learning (EBL) [20, 26, 46, 48, 56, 63, 64, 69, 76]. EBL methods can generalize

* Present address: Artificial Intelligence Research Branch, NASA Ames Research Center, Sterling Federal Systems, Mail Stop 244-17, Moffett Field, CA 94035, U.S.A.

Artificial Intelligence **40** (1989) 63–118

0004-3702/89/$3.50

from a single example by analyzing *why* that example is an instance of the concept. The explanation identifies the relevant features of the example, which constitute *sufficient conditions* for describing the concept. Typically, the purpose of EBL is to produce a description of the concept that enables instances of the concept to be recognized efficiently. The power of EBL stems from its use of a domain theory to drive the analysis process. Thus, while inductive learning is data-intensive, EBL is knowledge-intensive. As such, EBL represents part of a more general trend in AI towards knowledge-based systems.

In this article we describe the general EBL approach to learning, and discuss issues that arise when using EBL to improve problem solving performance. In particular, we explore one method, implemented in the PRODIGY problem solving system, for learning search control knowledge from problem solving experience. By providing PRODIGY's learning component with axiomatized knowledge about the architecture of the problem solver itself, as well as the application domain, we can use EBL to implement strategies for optimizing problem solving performance. This knowledge-based approach gives PRODIGY the capability to improve its performance rapidly as it solves problems in a particular domain. In the late 1950s John McCarthy declared that, "Our ultimate objective is to make programs that learn from their experience as effectively as humans do" [15, p. 360]. The EBL paradigm takes a major step towards effective learning by using a domain theory to guide the learning process. PRODIGY takes the next step in that direction by employing EBL as a general method for improving problem solving performance.

In the next section, we review the EBL paradigm and present several examples from the literature. Next, we discuss the issues that arise when employing EBL to improve problem solving performance. We then describe in detail how PRODIGY learns control knowledge to improve its problem solving efficiency, and conclude by comparing PRODIGY with other EBL problem solving systems.

2. The EBL Paradigm

The development of explanation-based learning has been a collaborative, evolutionary effort, marked by a gradual transition from exploratory research to more general and well-defined methods. The roots of EBL can be traced back to early analytical learning programs, such as STRIPS [29], HACKER [79] and Waterman's poker player [86], that improved their performance on the basis of experience. These EBL precursors, developed before the phrase was coined, did not investigate the method systematically. Thus the "modern" EBL era did not start until the early 1980s, when a number of researchers including DeJong [19], Silver [75], Mitchell [54], Carbonell [10], and others [2, 71, 87], were independently working on projects in a variety of different domains where learning depended on analyzing why observed examples had some significant property. These projects all emphasized knowledge-based learning

from a single example, in contrast to much of the research on inductive learning which was being conducted at the time. Eventually a series of comparison papers [20, 48, 56, 58, 69] were written that attempted to unify these approaches in a single paradigm. The term *explanation-based learning*, which was suggested by DeJong, has come to be identified with the paradigm. To illustrate EBL and its progression we will describe four examples demonstrating different approaches to EBL, as well as some of the applications that have been proposed.

2.1. The STRIPS approach

The STRIPS macro-operator formation technique [29] is perhaps the most influential precursor of present EBL techniques. After the STRIPS planner solves a problem, the plan can be turned into a set of macro-operators for solving similar problems in the future. In the STRIPS task domain a robot can move from room to room and push boxes together. Consider the problem of achieving (NEXT-TO BOX1 BOX2) when the robot is in ROOM1 and BOX1 and BOX2 are in an adjacent room, ROOM2. One plan that STRIPS might construct to solve this problem is:

```
(GOTO-DOOR ROOM1 ROOM2)
(GOTHRU-DOOR ROOM1 ROOM2)
(GOTO-BOX BOX1)
(PUSH-BOX BOX1 BOX2)
```

By analyzing why the plan solved the problem, the STRIPS macro-operator learning method produces a general plan for achieving any goal matching (NEXT-TO box-x box-y):

```
(GOTO-DOOR rm-w rm-v)
(GOTHRU-DOOR rm-w rm-v)
(GOTO-BOX box-x)
(PUSH-BOX box-x box-y)
```

The procedure for generalizing the plan involves more than simply replacing constants by variables. STRIPS analyzes why each step in the plan is necessary so that, for example, two identical constants can be replaced by distinct variables if doing so does not disturb the structural integrity of the plan. Once the plan is generalized, the preconditions of the resulting macro-operator describe conditions under which the goal (NEXT-TO box-x box-y) is solvable.

The EBL perspective on this process is that the successful plan explains why the goal (NEXT-TO BOX1 BOX2) is achievable. The general concept of interest is the class of situations in which (NEXT-TO box-x box-y) is achievable. Macro-operator formation computes sufficient conditions for membership in this class,

represented by the preconditions of the macro-operator. Thus, in contrast to inductive techniques, the planner learns from single examples.

2.2. Explanatory schema acquisition

More recently, DeJong and his students have experimented with a series of EBL programs that learn schemata for tasks such as problem solving and natural language understanding [19–21, 57, 63, 72, 74]. For instance, the GENESIS system [57, 59] is a schema-based natural language understanding system that learns new schemata in the normal course of processing narratives. Consider the first kidnapping story in Fig. 1. Although GENESIS may not know anything about kidnappings per se, the story is detailed enough so that GENESIS can understand the important events in the story in terms of pre-existing low-level schemata such as CAPTURE and BARGAIN. Thus the system can build up a causal explanation that relates the goals of the characters and the actions in the story. The explanation can then be generalized by eliminating details incidental to the explanation, such as the names of the particular characters involved. The resulting schema reflects only the constraints necessary to preserve the causal structure of the explanation, and serves as a general schema for understanding kidnappings.

Using the learned schema, GENESIS is able to understand sketchy stories such as the second story shown in Fig. 1. This sketchy story could not have been understood by the system had it not seen the first, causally complete, kidnapping story. Notice that the sketchy story leaves out information such as why Bob imprisoned Alice. Without the learned schema, the inferencing necessary to understand the sketchy story is combinatorially explosive. The

Story1:
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 Treno then John would release Mary. Fred gave him the money and John released Mary.

Story2:
Ted is the husband of Alice. He won $100,000 in the lottery. Bob imprisoned Alice in his basement. Bob got $75,000 and released Alice.

Fig. 1. Two kidnapping stories.

learned schema thus enables GENESIS to connect the events in the sketchy story without incurring the cost of the combinatorial explosion.

2.3. Constraint-based generalization

A third example of EBL is due to Minton [48], who describes a game playing program that learns by analyzing why its opponent was able to force it into a trap. Game playing offers a fertile domain for EBL because games are well-defined, so that clear-cut explanations of why a move was appropriate can often be found [3, 80, 83]. Consider the chess diagram in Fig. 2, which illustrates a simple chess combination called a "skewer." The black bishop has the white king in check. After the king moves out of check, as it must, the bishop can take the queen.

Minton's program can learn about tactical combinations of this type, where one opponent forces the other into an outright loss, a capture, or some other undesirable state. After falling into a trap, the program analyzes why the trap succeeded, and learns a rule that enables it to avoid the trap, or spring it on an opponent. The analysis operates by reconstructing the causal chain of events that forced the program into an undesirable state. In conducting this causal analysis, the program identifies a sequence of rules that explain why the trap succeeded. By computing the *weakest preconditions* of that sequence of rules the program can identify the general constraints that enable the trap to succeed. In our example, such an analysis can establish that while the pawns were irrelevant, the queen had to be "behind" the king for the plan to succeed. Ultimately, a general set of preconditions for applying this combination can be found. In future games this knowledge can be used to the system's advantage, both to avoid falling into traps of this type and to recognize when they can be employed against an opponent.

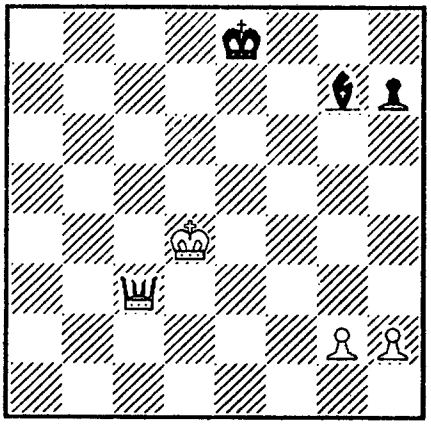

Fig. 2. A skewer position in chess.

2.4. The EBG approach

Mitchell, Keller and Kedar-Cabelli describe a unified approach to EBL, called explanation-based generalization (EBG) [56], which clarifies many of the common aspects of the various systems developed since STRIPS. One of the major contributions of the EBG approach is that explanations are identified with proofs, thus giving a precise meaning to the term "explanation" (as discussed by Minton [52]). Furthermore, Mitchell et al. suggest a clear specification of the input and output of EBL, as shown in Fig. 3. The input consists of a target concept,[1] a theory for constructing explanations, an example, and an operationality criterion that defines what it means for a description to be useful. Here *concepts* are viewed as predicates over instances, and therefore denote sets of instances. Thus a concept such as "BOX" refers to the set of objects that are identified as boxes. An *explanation* is a proof that the instance is a valid example of the concept. After generalizing the explanation, EBG produces an operational description that constitutes *sufficient conditions* for recognizing the target concept.

As an example, Mitchell et al. consider the target concept SAFE-TO-STACK(x, y), that is, the set of object pairs $\langle x, y \rangle$ such that x can be safely stacked on y. The target concept definition, training instance, theory, and operationality criterion are given in Fig. 4.

Given:

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

Determine:

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

Fig. 3. Mitchell et al.'s specification of EBL.

[1] We have slightly modified Mitchell et al.'s terminology for consistency with the remainder of this paper. In particular, we use the term *target concept* rather than *goal concept*, since the latter could be confused with the goals of the problem solver.

Target concept definition:
SAFE-TO-STACK(x, y) iff NOT(FRAGILE(y)) or LIGHTER(x,y)

Training example:
ON(OBJ1, OBJ2)
ISA(OBJ1, BOX)
ISA(OBJ2, ENDTABLE)
COLOR(OBJ1, RED)
COLOR(OBJ2, BLUE)
VOLUME(OBJ1,1)
DENSITY(OBJ1, 0.1)

Domain theory:
VOLUME(p_1,v_1) and DENSITY(p_1,d_1)→WEIGHT(p_1,$v_1 * d_1$)
WEIGHT(p_1,w_1) and WEIGHT(p_2,w_2) and LESS(w_1,w_2)
→LIGHTER($p1$,$p2$)
ISA(p_1, ENDTABLE)→WEIGHT(p_1, 5) [default]
LESS(1, 5)

Operationality criterion:
The learned concept description must be expressed in terms of the predicates used to describe examples (e.g., VOLUME, COLOR, DENSITY) or other selected, easily evaluated predicates from the domain theory (e.g., LESS).

Fig. 4. The SAFE-TO-STACK example for EBG.

The definition of SAFE-TO-STACK specifies that an object can be safely stacked on a second object if the second object is not fragile or the first object is lighter than the second. The domain theory encapsulates the system's knowledge about objects, weight, etc. The training example illustrates an instance of two objects, OBJ1 and OBJ2, that can be safely stacked on top of each other. Finally, the system's operationality criterion specifies that the explanation must be expressed in terms of easily evaluated predicates.

Now we are in a position to describe how EBG learns from this example. EBG proves that OBJ1 is SAFE-TO-STACK on OBJ2 (see Fig. 5). The proof is then generalized by *regressing* the target concept through the proof structure. The regression process replaces constants with variables while preserving the structure of the proof. The purpose of regression is to find the weakest conditions under which the proof structure will hold. In this manner EBG produces the following sufficient conditions for describing the concept SAFE-TO-STACK:

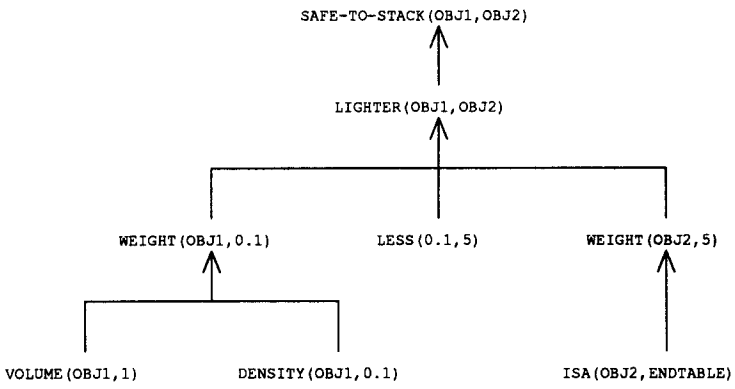

Fig. 5. An EBG proof tree.

VOLUME(x, v_1) and
DENSITY(x, d_1) and
LESS($v_1 * d_1$, 5) and
ISA(y, ENDTABLE)

These conditions specify that x can be safely stacked on y if y is an endtable and the volume times the density of x is less than 5. (Notice that the domain theory specifies that all endtables weigh 5 lbs by default.) This description satisfies the operationality criterion and is justified by the proof.

2.5. Discussion

The four examples of EBL that we have reviewed share a common, fundamental theme. In each case, there was a general target concept that the system attempted to learn: STRIPS learned conditions under which two boxes could be placed NEXT-TO each other, GENESIS learned how a kidnapping could succeed in meeting a character's goals, Minton's game playing program learned how a skewer could succeed, and the EBG method was used to learn the conditions under which an object could be safely stacked on another object. As we have seen, the power of these programs stems from their ability to construct an explanation after observing a single example. The explanation answers the question: Why is the training example an instance of the target concept? Thus, computing the weakest conditions under which the explanation holds produces sufficient conditions for recognizing the target concept.

A question commonly asked about EBL is: In what sense does the system learn if it starts with a definition of the concept? Simon has defined learning as "any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population" [77]. The definition of the concept that the system receives as input is nonoperational, which means that it is expensive (if not impossible) to use it to

recognize instances of the concept. EBL can be viewed as operationalizing the definition—transforming it to a definition that can be used as an efficient recognizer for the concept. The operationality criterion enforces the efficiency requirement on the sufficient conditions that are the output of EBL.

Another important question is: What role does the example play in EBL? In principle, given a complete domain theory the concept's definition could be operationalized without the example [39, 61]. However, the example serves two functions. First, the example guides the search for an operational definition thereby making the search more tractable. Second, to the extent that the example is representative of the instances of the concept that the system will be asked to recognize, the example leads to recognizers (sufficient conditions) that are likely to be applicable.

2.6. Current issues

EBL is still a rather new paradigm and an area of very active research in machine learning [22]. The specification of EBL described previously is only an initial specification of the EBL problem. Several attempts have been made to refine EBL's formalization. Furthermore, in extending the use of EBL a host of new issues arise. The following is a brief survey of some of the important issues currently being explored.

2.6.1. *Formalizing EBL*

Work on domain-independent EBL algorithms [20, 56] and on EBL in logic environments [34, 36] has pointed to the intimate relation between EBL on the one hand, and logic programming and theorem proving on the other [6, 53]. Indeed, recent papers by van Harmelen and Bundy [84] and Prieditis [67] argue that EBL algorithms are essentially equivalent to partial evaluation algorithms. However, van Harmelen and Bundy discount the importance of training examples for EBL. In doing so they largely overlook the fact that EBL is intended as a technique for learning from experience. The example plays an essential role in EBL by allowing a domain-independent problem solver to adapt to the particular environment it finds itself in. An analysis that attempts to clarify the role of examples in EBL, and the utility of EBL for improving problem solving performance appears in [47].

2.6.2. *The domain theory*

Making explicit the role of the domain theory in the generalization process led Mitchell et al. [56] to consider how EBL would perform given an imperfect domain theory. They described three types of imperfect theories that have since received considerable attention [68]:

– When the theory is *incomplete* it may be impossible to construct an explanation due to a lack of information. However, generating partial or

plausible explanations may suggest experiments or hypotheses useful for strengthening the theory [12, 32, 68].

– When the theory is *intractable*, an explanation may exist in principle, but finding it would take too long in practice. For example, the rules of chess form a complete theory of the game, but trying to prove that P-K4 belongs to the class of winning moves might take longer than the lifetime of the universe. Methods for moving from an intractable theory to a tractable, but approximate theory are being explored by a number of researchers, including Tadepalli [80] Bennett [4], Chien [14], Ellman [27], and Braverman and Russell [7].

– When the theory is *inconsistent* a desired assertion may be provable, but its negation will also be provable. This problem can arise when a theory has default rules that allow two inconsistent explanations to be formulated. In general, default or nonmonotonic logics are useful for generating explanations in the presence of uncertainty, but may necessitate retracting learned information. This problem may also arise if the theory is simply wrong, in which case experiments may help to resolve the inconsistency. Doyle [25] reports on a failure-driven approach for refining explanations constructed from an inconsistent theory.

The role of the domain theory in justifying explanation-based generalizations is another important issue. Mitchell et. al. argued that the domain theory enables EBL to produce generalizations that are deductively justified in terms of the learner's domain knowledge, whereas an inductive learner inevitably relies on some form of inductive bias to guide its generalization. Consequently, Mitchell et al. suggest that EBL techniques "provide a more reliable means of generalization" than inductive techniques [56, p. 48]. In response, Etzioni [28] argues that inductively formed generalizations may be justified by testing them on small samples. The test may be crafted to guarantee arbitrarily reliable generalization. Consequently, although the domain theory provides EBL with a measure of justification (a measure that is only as dependable as the theory), inductive learning may be *test-justified* to guarantee reliable generalization as well.

Furthermore, although EBL systems make no inductive leaps, bias plays a role in EBL as well. A learning system's bias determines which of the generalizations consistent with its training data is chosen [55]. Since an EBL system learns from a single example using a domain theory, the bias is encoded in the domain theory; the axioms that are used to construct a proof determine the generality of the concept description that results from the proof. Therefore, it is important to have a theory that will produce descriptions that are both general and useful (e.g., efficient for recognition). One important open question is how to build programs that can reason about and modify their own biases. Utgoff [83] has considered automatic methods for adjusting the bias in LEX, an inductive learning program. Related methods may also be useful for EBL.

2.6.3. *The explanation*

Significant effort has gone into crafting EBL's proofs or explanations to achieve superior generalizations. For example, several researchers [13, 16, 66, 74] have shown that, in some domains, producing descriptions at a useful level of generality may require explanations specifically suited to repetitive events. In the blocks world, for instance, to explain how to build towers of arbitrary height, it is useful to capitalize on the repetitive nature of the task by generalizing the number of blocks that appear in any given example. This is referred to as "generalization-to-N" [74]. One can view this body of work as investigating alternative *generalization* algorithms for EBL. However, we view this research and related efforts as investigating alternative *explanation* algorithms for EBL. This perspective was suggested by Minton [52, 53], who argued that the generalization process in EBL can always be described as computing the weakest preconditions (or equivalently, weakest premises) of an explanation. This provides a simple formal model of EBL, in which the semantics of the generalization process is fixed and only the explanation process varies.

2.6.4. *The example*

Although this notion has yet to be made precise, it appears intuitively clear that the example used in EBL focuses the learning mechanism by constraining the explanation process. However, Keller [38] and Mostow [61] have considered EBL-like systems that can search for operational concept descriptions without the benefit of examples to constrain the search. Another possibility that has received considerable attention is combining EBL with inductive techniques that make use of *many* examples. Utilizing both a domain theory and multiple examples appears to be a promising approach to addressing the problems raised by imperfect theories. For example, Lebowitz has explored combining inductive methods and explanation-based methods in the context of his UNIMEM system [44, 45]. Flann and Dietterich [30] have explored using explanations as input to an inductive component. Distinct approaches to this problem have been taken in [5, 16, 17, 65].

2.6.5. *The operationality criterion*

As pointed out by DeJong and Mooney [20], the EBG paper [56] suggests some very simple operationality criteria that are not necessarily realistic. In practice, it is difficult to guarantee that the knowledge learned by EBL will be useful. In some cases, the benefits of applying the knowledge may compare unfavorably with the costs of testing whether the knowledge is applicable [51]. Both Minton [52] and DeJong and Mooney [20] describe techniques for reformulating explanations to increase the utility of learned knowledge. In some cases it may be necessary to trade efficiency for generality, as pointed out

by Segre [73], because knowledge that is expressed in a very general way may be more expensive to employ. Keller [40] considers the operationality question in depth, comparing several different operationality criteria that have been used in EBL systems. Finally, Hirsh [34, 35] suggests explicitly reasoning about operationality.

2.6.6. *The target concept*

The question of "what to learn" has only just begun to be addressed in EBL applications [37, 39]. For example, EBL problem solvers have primarily been used to learn from successful operator sequences. Recently, however, EBL was demonstrated to be useful for learning from problem solving failures as well [14, 31, 33, 50, 62]. In Section 5 of this paper we explore this issue further and describe how PRODIGY uses multiple target concepts, each of which corresponds to a distinct strategy for improving problem solving performance (i.e., replicating success, avoiding failure, coping with goal interactions). This approach achieves much better results than if the system is confined to a single strategy, such as learning from successful operator sequences.

3. EBL as a Method for Improving Problem Solving Performance

Let us now consider in greater depth the issues involved in using EBL to improve problem solving performance. In the past, EBL problem solvers learned primarily by observing solutions to problems, and ignoring failure paths or other problematical impasses. However, learning from successful solutions is only one stategy for improving performance, and EBL can be used to learn a general class of optimization strategies.

First, let us be precise about our terminology. An optimization strategy is simply a method for improving a program with respect to some performance metric. Optimization strategies may be heuristic, in that they may typically improve performance, but not be guaranteed to do so in every instance. We note that, theoretically, the problem of improving a general problem solver so that it is optimal with respect to efficiency (or some other arbitrary performance metric) is undecidable. In any event our emphasis is on *improved* performance, not necessarily optimal performance.

A *dynamic* optimization strategy modifies a program based on its observed behavior, in contrast to a *static* optimization strategy, which modifies a program in isolation. For example, an optimizing compiler typically performs only static optimizations, such as constant folding, loop unrolling, and extraction of loop invariants [1]. In contrast, STRIPS' macro-operator learning technique is a dynamic optimization strategy, because each macro-operator is

acquired by observing the solution to a problem. Here the problem solver operating in a particular domain constitutes the program, and the addition of the macro-operator to the set of operators constitutes an optimization for that domain.

EBL is well-suited for implementing dynamic optimization strategies because it provides a mechanism for improving problem solving behavior on the basis of observed examples. In this paper we focus on optimizations that can be expressed as control knowledge. Specifically, our optimization strategies will be represented by rules of the form IF TEST(*st*) THEN DO ACTION(*st*) where *st* is a state of the problem solver, TEST(*st*) is true if TEST matches state *st*, and ACTION(*st*) is some modification of the program's normal behavior in state *st*. By letting the TEST be a target concept, EBL can produce a specialization of the TEST that is efficient to evaluate, whereas the original TEST is presumably very inefficient to evaluate.

To illustrate our approach, we have implemented a variety of optimization strategies in the PRODIGY problem solving system. Currently the system can learn by observing problem solving failures, successes, and interfering goals, each of which corresponds to a target concept. Furthermore, the set of target concepts is specified declaratively so that additional optimization strategies can be implemented.

Before describing our approach in detail, let us consider the issues that arise when using EBL to implement dynamic optimization strategies. For clarity, the following list is organized according to Mitchell et al.'s description of EBL's inputs:

– *Target concept coverage.* The learning component's target concepts describe the conditions under which each optimization strategy is applicable. Thus the coverage of the target concepts determines the situations where performance can be improved.

– *Scope of the theory.* The theory provides the means for proving that an optimization strategy applies in a given problem solving episode. The scope of the theory delimits the types of explanations that the system can construct. To implement dynamic optimization strategies, the theory must be capable of explaining relevant aspects of the system's problem solving behavior.

– *Method of constructing explanations from examples.* To learn by observing a problem solver, the system must be able to identify examples of the target concepts from the problem solving trace, and efficiently explain why the examples are subsumed by the target concepts. This requires a means for mapping from the trace to an explanation.

– *Operationality criterion.* The system's operationality criterion must reflect the computational costs and benefits of the transformations that are learned. In particular, the operationality criterion must insure that the time cost of testing

whether a transformation is appropriate does not outweigh the benefits provided by the transformation.[2]

Learning does not happen in a vacuum. In practice, the design of the problem solving architecture will greatly affect the design of the learning system, and the optimization strategies that can be implemented. Therefore, when designing the problem solving architecture, the following issues also need to be considered:

– *Flexibility*. Learning control knowledge is useful only if the problem solver is sufficiently flexible to take advantage of the learned knowledge. The language used to encode the problem solver's control knowledge will determine what the EBL component can express to the problem solver.
– *Parsimony*: The problem solver must be relatively simple. In general, the simpler the problem solver, the easier it is to formulate a theory describing its behavior.

There is a synergistic relationship between the problem solver and the EBL subsystem in PRODIGY. The problem solver provides problem solving episodes for the EBL component to analyze, and is a testbed for measuring the utility of the learned control knowledge. In turn the EBL component provides the problem solver with control knowledge.[3] Due to the power of EBL, sophisticated general search control strategies such as least commitment to not necessarily need to be built into the problem solver. Instead, appropriate domain-specific search control knowledge can be acquired via learning.

4. The PRODIGY System

PRODIGY is a domain-independent problem solver that acquires new knowledge by analyzing its experiences and interacting with an expert. In addition to the EBL method which this paper focuses on, there are several other learning components (described below). In essence, the PRODIGY architecture is inspired by observation of how human students transition gradually from novice to increasingly more expert performance by being taught through problem solving practice and much trial and error. Although not a fine-grain cognitive model, PRODIGY nevertheless strives to retain the human flexibility of applying focused expertise when available, and relying on less focused general problem solving behavior when domain knowledge fails to produce an adequate answer.

[2] Learning is typically used to improve performance by reducing search time, but other performance benefits might additionally be realized. For example, qualitatively better plans can be produced. We assume that there is a single metric for comparing the costs and benefits of learned knowledge.

[3] The EBL component may also direct the problem solver to explore alternative solutions so that EBL can generate a proof of some conjecture.

Thus, the learning components acquire either factual domain knowledge or domain-specific control knowledge, both necessary components for search-limited, knowledge-intensive, expert behavior. This new knowledge is then used by the problem solver (whose structure does not change) when encountering new, progressively more difficult problems in the same domain.

PRODIGY consists of a general means-ends problem solver connected to a set of learning modules: EBL, derivational analogy, plan abstraction, and experimentation. These modules augment the domain-specific knowledge bases, including operators and control rules, through incremental problem solving experience. The PRODIGY architecture is diagrammed and discussed further in Appendix A and the reader is referred to the references cited therein for in-depth discussion of the various learning modules. This article, however, focuses only on EBL and the necessary support to EBL provided by the problem solving engine.

4.1. The problem solver

In order to solve problems in a particular domain, PRODIGY must first be given a specification of that domain, which consists of a set of operators and inference rules. Operators correspond to external actions with consequences in the world. Each operator has a precondition expression that must be satisfied before the operator can be applied, and a list of effects that describes how the application of the operator changes the current state of the world. The effects can either directly or conditionally add or delete atomic formulas from the state description.

Inference rules, unlike operators, do not correspond to external actions; they simply increase PRODIGY's explicit knowledge about the current state. Even so, inference rules are specified much in the same way that operators are. Each inference rule has a precondition expression that must be true for the rule to be applicable. Each inference rule also has a list of effects. However, inference rules only add formulas to the state description; they never delete formulas. Because inference rules are treated similarly to operators, PRODIGY can use a homogeneous control structure, enabling the search control knowledge to guide the application of operators and inference rules alike. The homogeneous control structure makes for a parsimonious problem solver.

There are two types of predicates used in the system: primitive predicates and defined predicates. Primitive predicates are directly observable, or *closed-world* [15, p. 115], and they may be added to and deleted from the state by operators. Defined predicates are inferred (on demand) using inference rules. They represent useful abstractions in the domain, allowing operator preconditions to be expressed more concisely.

PRODIGY's description language (PDL), is a form of predicate logic that allows negation, conjunction, disjunction and existential quantification, as well

as universal quantification over sets.[4] Preconditions of operators, inference rules, and also search control rules (see Section 4.3) are expressed in PDL.

4.2. The problem solving process

To facilitate our discussion, we will describe PRODIGY in terms of a simple machine-shop scheduling task. The shop contains several machines, including a lathe and a roller that are used to reshape objects, and a polisher. We assume that each machining operation takes one time unit. Given a set of objects to be polished, shaped, etc., and a finite amount of time, the task is to schedule the objects on the machines so as to meet these requirements. (Note that we are considering a satisficing rather than an optimizing task.) The specifications for the LATHE, ROLL, and POLISH operators are shown in Appendix B. (A complete specification of the domain can be found in [52].)

Consider the LATHE operator:

```
(LATHE (obj time)
 (PRECONDITIONS
   (AND
     (LAST-SCHEDULED obj prev-time)
     (LATER time prev-time)
     (NOT (EXISTS other-obj SUCH-THAT
            (SCHEDULED other-obj LATHE time)))))
 (EFFECTS
   (DELETE (SHAPE obj old-shape))
   (DELETE (LAST-SCHEDULED obj prev-time))
   (IF (POLISHED obj)
       (DELETE (POLISHED obj)))
   (ADD (LAST-SCHEDULED obj time))
   (ADD (SHAPE obj CYLINDRICAL))
   (ADD (SCHEDULED obj LATHE time))))
```

The LAST-SCHEDULED relation indicates the time period at which an object was last scheduled to be operated on. Thus, the first two preconditions force the problem solver to schedule operations chronologically; once an object has been scheduled for a machine at time t, it cannot be scheduled for another operation at an earlier time period. (Initially, all objects are LAST-SCHEDULED at TIME-0.) This restriction will make our examples easier to understand, and will enable us to represent properties of objects in a straightforward manner. (Inefficiencies due to this restriction can be largely compensated for by control knowledge.)

[4] Constants are shown in upper case and variables are in italics. To enhance readability, quantifiers are omitted when they are obvious. The reader may note that PDL is more expressive than Horn clauses in PROLOG that permit neither disjunction nor explicit quantification over sets.

The last precondition of LATHE states that there cannot be another object scheduled to be lathed during the same time period. The effects of the operator include scheduling the object on the LATHE, and updating the LAST-SCHEDULED relation. Furthermore, the effects indicate that lathing changes the object's shape, and removes the polish from its surface (if it is polished).

The ROLL and POLISH operators have similar preconditions and effects. Notice, however, that polishing an object requires that the object must either be rectangular, or clampable to the polisher. The CLAMPABLE predicate is a defined predicate; the domain specification in Appendix B includes an inference rule for determining whether an object is clampable to a given machine. All other predicates referred to in this paper are primitives (i.e., closed-world).

Once the domain has been specified, problems are presented to the problem solver by describing an initial state and a goal expression to be satisfied. The goal expression for our example problem is:

```
(AND (SHAPE OBJECT-A CYLINDRICAL)
     (POLISHED OBJECT-A))
```

This expression is satisfied if there is an object named OBJECT-A that is polished and has a cylindrical shape.[5]

The initial state for our example is illustrated in Fig. 6. OBJECT-B and OBJECT-C have already been scheduled, while OBJECT-A has yet to be scheduled. Let us suppose that the schedule consists of 20 time slots, and that OBJECT-A is initially unpolished, oblong-shaped, and cool.

The search tree for this example is shown in Fig. 7. The left side of each node shows the goal stack and the pending operators at that point. The right side shows a subset of the state that is relevant to our discussion. For example, at Node 3, the current goal is to make the object CLAMPABLE, a precondition

	TIME-1	TIME-2	TIME-3	TIME-4....				....TIME-20
LATHE	OBJECT-B							
ROLLER	OBJECT-C							
POLISHER		OBJECT-B						

Fig. 6. The initial state provided to PRODIGY.

[5] We have simplified our representation so that attributes such as shape and temperature take ordinal values such as RECTANGULAR and COOL. Normally, a richer representation would be used to reflect real-world complexities. For example, a shape might be represented by the list [CYLINDRICAL 1 3], describing a cylindrical shape with diameter 1 and length 3. When necessary, predicates such as IS-LENGTH can then be used to extract particular fields of lists. For example the following formula would be true, only if x equals 3: (IS-LENGTH x [CYLINDRICAL 1 3]). This is easy to implement because PRODIGY allows *static* predicates (those whose truth value is not dependent on the state), such as IS-LENGTH and LESS-THAN, to be computed by user-defined functions.

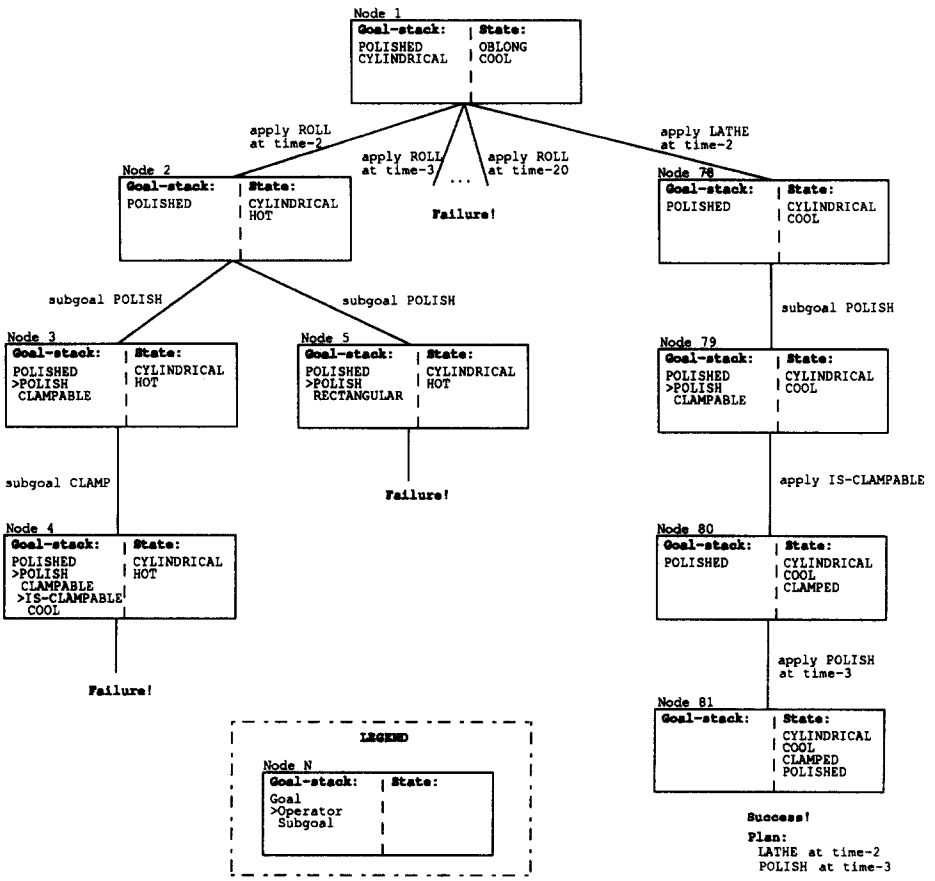

Fig. 7. PRODIGY's search tree prior to learning.

of the POLISH operator that is being considered to achieve the higher goal of being polished. The predicates like CYLINDRICAL and HOT in the figure are shorthand for the actual formulas, such as (SHAPE OBJECT-A CYLINDRICAL) and (TEMPERATURE OBJECT-A HOT).

By default, PRODIGY achieves goals from left to right in an expression, unless control knowledge indicates a more appropriate ordering. For each goal the system selects a relevant operator (or inference rule), matches the preconditions against the database, and subgoals if the match does not succeed. If the match does succeed, the operator (or inference rule) is applied to the current state, and then the system returns its focus to goals that remain to be achieved. In our example, (SHAPE OBJECT-A CYLINDRICAL) is not true in the initial state, therefore PRODIGY considers the operators LATHE and ROLL, since only these operators add effects that unify with this goal. At this point

PRODIGY arbitrarily decides to try ROLL first, as there are no control rules that indicate otherwise.

In order to satisfy the preconditions of ROLL, PRODIGY must select a time slot later than the last time slot for which OBJECT-A has last been scheduled (i.e., TIME-0). Furthermore, the machine must be idle at this time. Let us assume that previously acquired control knowledge indicates a preference for the earliest time slot that satisfies these constraints, TIME-2. After rolling the object at TIME-2, PRODIGY attempts to polish the object, but the preconditions of POLISH specify that the object must either be rectangular, or clampable to the polisher. Unfortunately, the object is not clampable because rolling the object has raised its temperature so that it is too hot to clamp. Furthermore, there is no way to make the object clampable, because there is no operator that will cool the object. And finally, since there is no way to make the object rectangular, the attempt to apply POLISH fails.

Backtracking, PRODIGY then tries rolling the object at TIME-3, and then TIME-4, and so on, until the end of the schedule is reached at TIME-20.[6] Each of these attempts fails to produce a solution, because rolling the object always makes it hot, and therefore unclampable. As we will see, when learning is interleaved with problem solving, PRODIGY can reason about the failures and therefore backtrack more intelligently. In any event, the problem solver finally succeeds when it eventually backs up and tries LATHING rather than ROLLING.

As our example illustrates, the problem solving process can be extremely expensive without control knowledge. The next section describes how control knowledge is represented in the PRODIGY architecture.

4.3. Control rules

The PRODIGY architecture separates domain knowledge, which specifies *what* operators and inference rules are available, from control knowledge, which describes *how* to solve problems in the domain. PRODIGY's control rules constrain the possible choices that the problem solver makes as it searches. The search is conducted by repeating the following decision cycle:

Step 1. A node in the search tree is chosen. A node consists of a set of goals and a state of the world.

Step 2. One of the goals at that node is chosen as the immediate focus. (The other goals will be addressed later, at subsequent nodes.)

Step 3. An operator relevant to fulfilling the goal is chosen.

Step 4. Bindings for the variables in the operator are selected. If the instantiated operator is applicable, then it is applied, otherwise PRODIGY subgoals on the operator's unmatched preconditions. In either case, a new node is created.

[6] PRODIGY does not have to consider alternative time slots for POLISH, because it has not yet selected one when it subgoals on CLAMPABLE.

At each of these choice points there is always a default set of candidates and a simple default decision strategy. For example, depth-first search is the default strategy for choosing a new node. There is a set of control rules for each of the four decisions (nodes, goals, operators, and variable bindings). Control rules modify the default behavior by specifying either that a particular candidate should be selected (over all others), rejected (from any future consideration), or that it should be preferred over another candidate or candidates (all else being equal).

Each control rule has left-hand side conditions testing applicability and a right-hand side indicating whether to SELECT, REJECT, or PREFER a candidate. To make a control decision, PRODIGY first applies the applicable selection rules to select a set of candidates. (If no selection rules are applicable, all the default candidates are selected.) Next, rejection rules further filter this set, and finally, preference rules are used to order the remaining alternatives.[7] If backtracking is necessary, the next most preferred is attempted, and so on, until all candidates are exhausted. (In effect, PRODIGY backtracks when it returns to a previously chosen node in order to explore a different goal, operator, and/or bindings.)

The control rule depicted in Fig. 8 is an operator rejection rule. Notice that the rule uses meta-level predicates such as ADJUNCT-GOAL and CURRENT-NODE in addition to domain-level predicates such as SHAPE. (An adjunct goal at a node will be achieved after the current goal.) The rule states that if the current goal at a node is to make an object cylindrical and the object must subsequently be polished, then the ROLL operator should be rejected.

The example problem from the previous section illustrates why this rule is appropriate: polishing OBJECT-A after rolling it turned out to be impossible. Had the problem solver previously learned this rule, the problem would have been solved directly, without the costly backtracking at Node 1. In general, control rules can serve the following purposes:

(1) *To increase the efficiency of the problem solver's search.* Control rules

```
IF (AND (CURRENT-NODE node)
        (CURRENT-GOAL node (SHAPE obj CYLINDRICAL))
        (CANDIDATE-OPERATOR node ROLL)
        (ADJUNCT-GOAL node (POLISH obj)))
THEN (REJECT OPERATOR ROLL)
```

Fig. 8. An operator rejection rule.

[7] Preferences are transitive. If there is a cycle in the preference graph, then those preferences in the cycle are disregarded. A candidate is "most preferred" if there is no candidate that is preferred over it.

guide the problem solver down the correct path so that solutions are found faster.

(2) *To improve the quality of the solutions that are found.* There is usually more than one solution to a problem, but only the first one that is found will be returned. By directing the problem solver's attention along a particular path, control rules can express preferences for solutions that are qualitatively better (e.g., more reliable, less costly to execute, etc.).

(3) *To direct the problem solver along paths that it would not explore otherwise.* As with most planners, for efficiency PRODIGY normally explores only a portion of the complete search space. This is accomplished through this use of default control rules. For example, PRODIGY will not normally explore all permutations of conjunctive sets of subgoals. However, when these default heuristics prove too restrictive, they can be overridden by additional selection rules.

PRODIGY's reliance on explicit control rules, which are typically learned for specific domains (e.g., by the EBL process), distinguishes it from most domain-independent problem solvers. Instead of using a least-commitment search strategy, for example, PRODIGY expects that any important decisions will be guided by appropriate domain-specific control rules. If no control rules are relevant to a decision, then PRODIGY makes an arbitrary choice. If in fact the wrong choice was made, and costly backtracking is necessary, an attempt will be made to learn the control knowledge that must be missing. However, should the choice prove immaterial no effort will have been wasted on the decision-making process. Thus, the rationale for PRODIGY's "casual commitment" strategy is that control rules are expected to guide any decision with significant ramifications. Instead of relying on sophisticated, but potentially very weak, domain-independent problem solving strategies, PRODIGY relies on learning to generate clever behavior. In fact, the more clever the underlying problem solver is, the more difficult the job will be for the learning component. For instance, the inclusion of complex domain-independent conflict resolution criteria [8] would make it difficult to add far more useful and constraining domain-specific control knowledge. In the next section, we describe in detail how the EBL learning component acquires control rules.

5. The EBL Component

In this section we discuss in detail PRODIGY's explanation-based learning component. As mentioned in Section 3, four basic design issues need to be addressed in integrating an explanation-based learning system with a problem solver. First, there is the question of *target concept coverage*. What are the system's target concepts? The second issue in the *scope of the theory*, which determines the space of explanations that the system can generate for a given target concept. The third is the *method for mapping from an example to an*

explanation. Is the explanation process expensive, or can the explanation be constructed directly from a problem solving trace? Finally, the last issue concerns the *operationality criterion*. What makes an explanation useful? How does the system determine whether a particular explanation satisfies this criterion? In the next four sections we discuss how PRODIGY addresses these issues, and in the following section we present several detailed examples of learning in PRODIGY.

5.1. Target concept coverage

Currently, for each of the four types of search control decisions that PRODIGY makes (i.e., choosing a node, goal, operator, or bindings, as described in Section 4.3), there are four types of target concepts: SUCCEEDS, FAILS, SOLE-ALTERNATIVE, and GOAL-INTERFERENCE. A control choice is said to *succeed* if it leads to a solution. Similarly, a choice *fails* if there is no solution to be found along that path. A choice is a *sole alternative* if all other alternatives fail. Finally, a choice results in *goal interference* if either a previously achieved goal is undone, or a precondition of a subsequent action in the plan is undone, even if (suboptimal) success is eventually achieved. Each type of target concept is associated with a control rule type, as given by Fig. 9.

The target concepts that we have selected for PRODIGY are useful and effective (as demonstrated below), but do not represent the complete range of potential target concepts. The language provided for specifying target concepts makes it easy to add target concepts as they are needed. For example, in some planning domains we believe it would be worthwhile to learn from advantageous goal interactions; thus far PRODIGY only learns to optimize against harmful interactions.

5.2. The scope of the theory

PRODIGY constructs search control rules by explaining why a target concept was satisfied by a training example—e.g., why a problem solving attempt failed, or why a goal interaction yielded a suboptimal plan. In order to construct explanations, we require a theory describing the relevant aspects of the problem solver, as well as a theory that describes the task domain (such as

Target concept type	Control rule type
Succeeds	Preference rule
Fails	Rejection rule
Sole alternative	Selection rule
Goal interference	Preference rule

Fig. 9. Correspondence of target concept type to control rule type.

our machine-shop world). Therefore, PRODIGY employs a set of *architecture-level* axioms that serve as its theory of the problem solver, and a set of *domain-level* axioms that are automatically derived from the domain specification.[8] Each axiom is a conditional statement written in PDL that describes when a concept (i.e., an atomic formula) is true. Figure 10 shows two architecture-level axioms and Fig. 11 shows two domain-level axioms.

The domain-level axioms state the domain operators' effects and preconditions. The architecture-level axioms describe how the problem solver operates. For example, the axioms for OPERATOR-SUCCEEDS shown in Fig. 10 state that an operator succeeds in solving a goal at a node if the operator directly solves the goal (i.e., the operator has an effect that unifies with the goal), or applying another operator results in a node at which the operator succeeds in solving the goal.

5.3. The mapping method

Mapping from a problem solving trace into an explanation involves first selecting an example of a target concept that is to be learned and then constructing an explanation of that particular example. We will first describe how examples are selected and then describe PRODIGY's explanation method.

5.3.1. *Selecting an example*

The EBL process can be either initiated after problem solving has terminated, or interleaved with problem solving. In either case, the learning component

---

Axiom-S1: An operator succeeds if it directly solves the goal.
(OPERATOR-SUCCEEDS *op goal node*)
if (AND (IS-EFFECT *goal op params*)
(APPLICABLE *op params node*))

Axiom-S2: An operator succeeds if it succeeds after precursor operator is applied
(OPERATOR-SUCCEEDS *op goal node*)
if (AND (APPLICABLE *next-op params node*)
(OPERATOR-SUCCEEDS *op goal child-node*)
(CHILD-NODE-AFTER-APPLYING-OP *child-node next-op node*))

Fig. 10. Architecture-level axioms describing OPERATOR-SUCCEEDS.

[8] The architecture-level axioms are domain-independent but must be hand-coded. In contrast, the domain-level axioms are created automatically for each domain by a simple conversion procedure that examines the operators, inference rules and control rules for the domain.

```
Axiom-D1: POLISH is applicable if its preconditions are satisfied.
(APPLICABLE op params node)
if (AND (MATCHES op POLISH)
        (MATCHES params (obj time))
        (KNOWN node (AND (OR (SHAPE obj RECTANGULAR)
                             (CLAMPABLE obj POLISHER))
                         (LAST-SCHEDULED obj prev-time)
                         (LATER time prev-time)
                         (NOT (EXISTS other-obj SUCH-THAT
                                 (SCHEDULED other-obj
                                   POLISHER time))))))

Axiom-D2: After applying the operator POLISH to an object it is polished.
(IS-EFFECT effect op params)
if (AND (MATCHES op POLISH)
        (MATCHES params (obj time))
        (MATCHES effect (POLISHED obj)))
```

Fig. 11. Examples of domain-level axioms for machine-shop scheduling.

begins by examining the explored portion of the search tree in order to find examples of its target concepts. After each suitable training example is selected, PRODIGY constructs an explanation, and from this explanation creates a search control rule. The system continues its examination until a fixed time limit is exceeded or the search tree has been fully analyzed.

PRODIGY conducts its examination of the tree in postorder: children are analyzed before parents, from left to right in the search tree. This bottom-up style of processing is preferred because, due to the fixed time limit, PRODIGY may not have the opportunity to analyze the entire tree. (The search tree may consist of thousands of nodes.) Beginning the analysis at the farthest limbs of the tree insures that some learning will take place even if the learning process must be aborted early due to time constraints. Thus the EBL process is incremental, similar in spirit to an "anytime algorithm" as defined by Dean and Boddy [18].

When PRODIGY examines a node in the tree, *training example selection heuristics* are used to select the examples that appear worth learning about. There is a set of selection heuristics associated with each target concept. For example, one selection heuristic for OPERATOR-SUCCEEDS, called "others-have-failed," states that a successful operator is interesting only if the problem solver previously tried an operator that failed. (If the problem solver immediately found the correct operator, there is no reason to learn additional control knowledge.) To illustrate this, consider the search tree from our machine-shop problem described in Section 4.2. In solving this problem, PRODIGY first tried

the ROLL operator, and only after ROLL failed did it try LATHE. Once LATHE was chosen, the system had no other difficulties with subsequent decisions and, for instance, chose the POLISH operator without incident. Thus, the success of LATHE is deemed interesting, but the subsequent success of POLISH is not.

However, even if an example is not interesting in its own right, it may be selected for explanation because the result can be used as a "lemma" in explaining a higher-level example. This is, in fact, the case with POLISH. The success of POLISH is not considered worth learning about. However, it is necessary to explain the success of POLISH in order to explain why LATHE was the appropriate choice higher up in the search tree. The explanation of POLISH's success will *not* be converted into a control rule.

It can also be the case that an example is both interesting in its own right, and useful for explaining higher-level examples. For instance, in order to explain why the ROLL operator failed at Node 1 it is necessary to explain the failure of Node 2, a child of Node 1. But this lower-level failure is also considered worth learning for its own sake. PRODIGY uses a selection heuristic for failure, called "highest-failure-per-node," which always considers node failures to be worth learning about. (Other failures, such as operator failures, are considered interesting only if the entire node did not fail.)

5.3.2. *Constructing an explanation*

PRODIGY constructs explanations using a method that we refer to as *explanation-based specialization* (EBS). A target concept is specialized by retrieving an axiom that describes the concept and recursively specializing the axiom, as described in Fig. 12. The recursion terminates upon encountering primitive formulas. The sequence of specializations proves that the example is a valid instance of the target concept.

When there is more than one axiom available to specialize a concept, as is the case with OPERATOR-SUCCEEDS, the system must select the axiom that corresponds to the training example. To find the appropriate axiom we allow each concept (i.e., each predicate) to be associated with a *discriminator* function that examines the search tree and selects an axiom corresponding to the example. Each discriminator function is a piece of code that takes the concept to be specialized and its example and returns the appropriate axiom with which to specialize the concept. (It also updates bookkeeping information associating the formulas in the axiom body with their corresponding examples in the search tree, so that the EBS algorithm can correctly specialize the axiom body.)

To illustrate how the EBS process operates, let us return to our machine-shop problem. As described in the previous section, the success of POLISH at Node 80 constitutes an example of OPERATOR-SUCCEEDS. Below we outline how the explanation for this example is constructed. (In Section 6.1 we show how this explanation is extended to explain why the LATHE operator preceding POLISH succeeded.)

To specialize a formula:

- If the formula is a conjunctive formula, (AND F_1 F_2...), then specialize each conjunct, and return the result: (AND (specialize F_1) (specialize F_2)...).
- If the formula is a disjunctive formula, (OR F_1 F_2...), then specialize the first disjunct consistent with the example, e.g. F_2, and return the result: (specialize F_2).
- If the formula is a universal formula, (FORALL (x_1...) SUCH-THAT F_1, F_2), then there will be a set of examples for F_2 in the search tree. For each of these subexamples, specialize F_2. Return the result:

  (FORALL (x_1...) SUCH-THAT F_1
    (OR (specialize F_2)$_1$
      (specialize F_2)$_2$...))

- If the formula is negated, (NOT F), then return the formula unchanged: (NOT F).
- If the formula is atomic, (P x_1 x_2...), and primitive (i.e., there are no axioms that define P) then return the formula unchanged: (P x_1 x_2...).
- If the formula is atomic, (P x_1 x_2...), and not primitive, then
  (1) Call the discriminator function associated with the concept P to retrieve an axiom that corresponds to the training example. The axiom will be a conditional of the form: (P y_1 y_2...) if axiom-body, where y_1, y_2, ... are distinct variables.
  (2) Replace the variables y_1, y_2, ... in the axiom with the corresponding values in the formula x_1, x_2, ...
  (3) Uniquely rename all other variables in the axiom body.
  Return the result: (specialize axiom-body).

Fig. 12. The EBS algorithm.

Target concept: (OPERATOR-SUCCEEDS *op goal node*)

Training example:
(OPERATOR-SUCCEEDS POLISH (POLISHED OBJECT-A) Node80)

To explain why the POLISH operator was successful, PRODIGY will specialize the concept OPERATOR-SUCCEEDS. In this case, since POLISH directly solved the goal, the system begins by specializing OPERATOR-SUCCEEDS with Axiom-S1 in Fig. 10, the axiom that corresponds to the training example. The system then recursively specializes the subconcepts APPLICABLE, and IS-EFFECT as shown below. The specializations performed at each step are indicated in bold face.

Target concept: (OPERATOR-SUCCEEDS *op goal node*)

Specialize (OPERATOR-SUCCEEDS *op goal node*) using Axiom-S1:
(AND (IS-EFFECT *goal op params*)
(APPLICABLE *op params node*))

Specialize (IS-EFFECT *goal op params*) using Axiom-D2:
(AND **(AND (MATCHES *op* POLISH)**
(MATCHES *params* (*obj-x time-x*))
(MATCHES *goal* (POLISHED *obj-x*)))
(APPLICABLE *op params node*))

Specialize (APPLICABLE *op params node*) using Axiom-D1:
(AND (AND (MATCHES *op* POLISH)
(MATCHES *params* (*obj-x time-x*))
(MATCHES *goal* (POLISHED *obj-x*)))
(AND (MATCHES *op* POLISH)
(MATCHES *params* (*obj-y time-y*))
(KNOWN *node* (AND (OR (SHAPE *obj-y* RECTANGULAR)
(CLAMPABLE *obj-y* POLISHER))
(LAST-SCHEDULED *obj-y time-z*)
(LATER *time-y time-z*)
(NOT (EXISTS *obj-w* SUCH-THAT
(SCHEDULED *obj-w* POLISHER
***time-y*)))))))**

Both APPLICABLE and IS-EFFECT can be specialized by more than one axiom. The appropriate specializations are determined by discriminator functions that retrieve the axioms corresponding to the training example.

As specialization proceeds, the system also simplifies the result in order to reduce its match cost. (This process, called *compression*, is described in [52].) Thus, after some trivial simplifications performed by the system, our specialized concept is re-expressed as follows:

(OPERATOR-SUCCEEDS *op goal node*)
if (AND (MATCHES *op* POLISH)
(MATCHES *goal* (POLISHED *obj*))
(KNOWN *node* (AND (OR (SHAPE *obj* RECTANGULAR)
(CLAMPABLE *obj* POLISHER))
(LAST-SCHEDULED *obj time-z*)
(LATER *time-x time-z*)
(NOT (EXISTS *obj-w* SUCH-THAT
(SCHEDULED *obj-w* POLISHER *time-x*))))))

The expression simply states that an operator succeeds in solving a goal at a node if the operator is POLISH, the goal is to have the object POLISHED, and the preconditions of POLISH are known to be satisfied at the node. (An expression is KNOWN at a node if it matches the state at that node.)

When the EBS process terminates we are left with a learned description that is a specialization of the target concept. The explanation that was used to arrive at this description is simply the sequence of specializations (and simplifications) employed by the EBS process; the learned description represents the weakest conditions under which the explanation holds. Many other EBL systems use a two-phase algorithm that first constructs the explanation and then finds its weakest preconditions (e.g., [48, 56, 58]). The EBS method computes the weakest preconditions in a single pass.

To improve the efficiency of the learning process, PRODIGY caches learned descriptions so that they can be used as lemmas in subsequent analyses. For example, in our machine-shop example the result describing why POLISH succeeded is cached and used to explain the success of LATHE. Similarly, the explanation of why Node 2 failed can not only be converted into a control rule, but also used as part of the explanation of why ROLL failed at Node 1. Moreover, cached results can prevent duplication of effort when two or more nodes fail for the same reason, since only one explanation has to be generated. This is described further in Section 6.2.

Once a description is returned by EBS, a search control rule can be constructed by filling in the *rule construction template* associated with the target concept. (Figure 13 shows the rule construction templates for OPERATOR-SUCCEEDS and OPERATOR-FAILS). This is simply a matter of replacing the target concept in the template with the learned description, and simplifying.

5.4. The operationality criterion

Learned control knowledge should not just be *usable*, but *useful* as well. Therefore PRODIGY extends the standard motion of operationality to include utility [61]. In other words, PRODIGY not only requires that learned control rules be executable, it also requires that they actually improve the system's performance. Specifically, the utility of a control rule is defined as the cumulative improvement in *search time* attributable to the rule. Utility is given by the cost-benefit formula:

$$\text{Utility} = (\text{AvrSavings} \times \text{ApplicFreq}) - \text{AvrMatchCost}$$

where

AvrMatchCost = is the average cost of matching the rule,
AvrSavings = the average savings when the rule is applicable,
ApplicFreq = the fraction of times that the rule is applicable when it is tested.

Target Concept: (OPERATOR-FAILS *op goal node*)

Training example selection heuristics: Highest-failure-per-node, . . .

Rule construction template:

```
IF (AND (CURRENT-NODE node)
        (CURRENT-GOAL node goal)
        (CANDIDATE-OPERATOR node op)
        (OPERATOR-FAILS op goal node))
THEN (REJECT OPERATOR op)
```

Target concept: (OPERATOR-SUCCEEDS *op goal node*)

Training example selection heuristics: Others-have-failed, . . .

Rule construction template:

```
IF (AND (CURRENT-NODE node)
        (CURRENT-GOAL node goal)
        (CANDIDATE-OPERATOR node op)
        (CANDIDATE-OPERATOR node other-op)
        (OPERATOR-SUCCEEDS op goal node))
THEN (PREFER OPERATOR op OVER other-op)
```

Fig. 13. Two target concept specifications.

After learning a control rule, PRODIGY maintains statistics on the rule's use during subsequent problem solving, in order to determine its utility. If the rule has negative utility, it is discarded.[9]

Because the representation of the left-hand side of a control rule significantly affects its match cost, the simplification process that occurs in conjunction with EBS can greatly increase the utility of the resulting rule. PRODIGY's utility evaluation process and the effect of simplification on utility are discussed by Minton [51, 52].

5.5. Results

Figure 14 summarizes the system's performance on one hundred randomly generated problems from the machine-shop domain. Three conditions are

[9] Unfortunately, although match cost and application frequency can be directly measured during subsequent problem solving, it is more difficult to measure the savings. Doing so would require running the problem solver with and without the rule on each problem. (This would have to be done for all rules.) Instead, PRODIGY uses an estimate of the rule's average savings based on the savings that the rule would have produced on the training example for which it was learned.

shown: the problem solver running without any control rules, the problem solver running with learned control rules, and problem solver running with hand-coded control rules. The learned control rules were acquired during a separate training phase consisting of approximately one hundred problems. The hand-coded control rules were written by the authors. (They took about eight hours to code.) The graph shows how the cumulative problem solving time grows as the number of problems solved increases. The cumulative time is the total problem solving time *over all problems* up to that point. Thus, the slopes of the curves are positive because the y-axis represents cumulative time. Because the problems were ordered according to size (i.e., number of objects to be scheduled, etc.), and therefore progressively more difficult, the second derivatives of the curves are also positive.

As the graph shows, PRODIGY performed approximately fifty percent worse with the learned rules than with the hand-coded rules, but much better than without control rules. Similar results were obtained in other domains as well. In other experiments, PRODIGY's EBL methods were also shown to perform better than standard macro-operator methods. A detailed discussion of these experimental results can be found in [52]. Overall, we have concluded that EBL appears to be a promising approach to performance optimization. However, we believe that methods for generating even better explanations must be found before PRODIGY's learned control rules equal hand-coded control rules in performance.

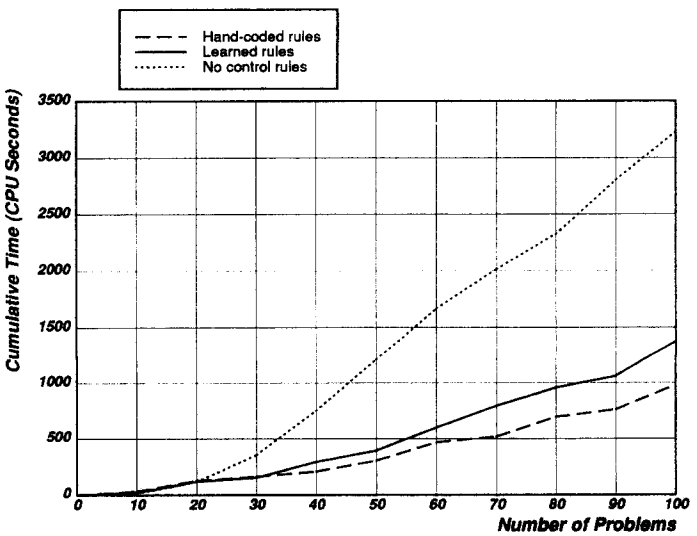

Fig. 14. Performance results from the machine-shop scheduling domain.

6. Examples of Learning in PRODIGY

Having described PRODIGY's EBL method, we now focus on concrete examples of learning from success, failure and goal interference. In each case, we present the relevant fragment of the pre-learning search tree and the steps in the subsequent analysis that yields new control rules.

6.1. Learning from success

Returning to the machine-shop domain, let us illustrate how PRODIGY learns from success. Figure 15 shows the nodes along the solution path (the right

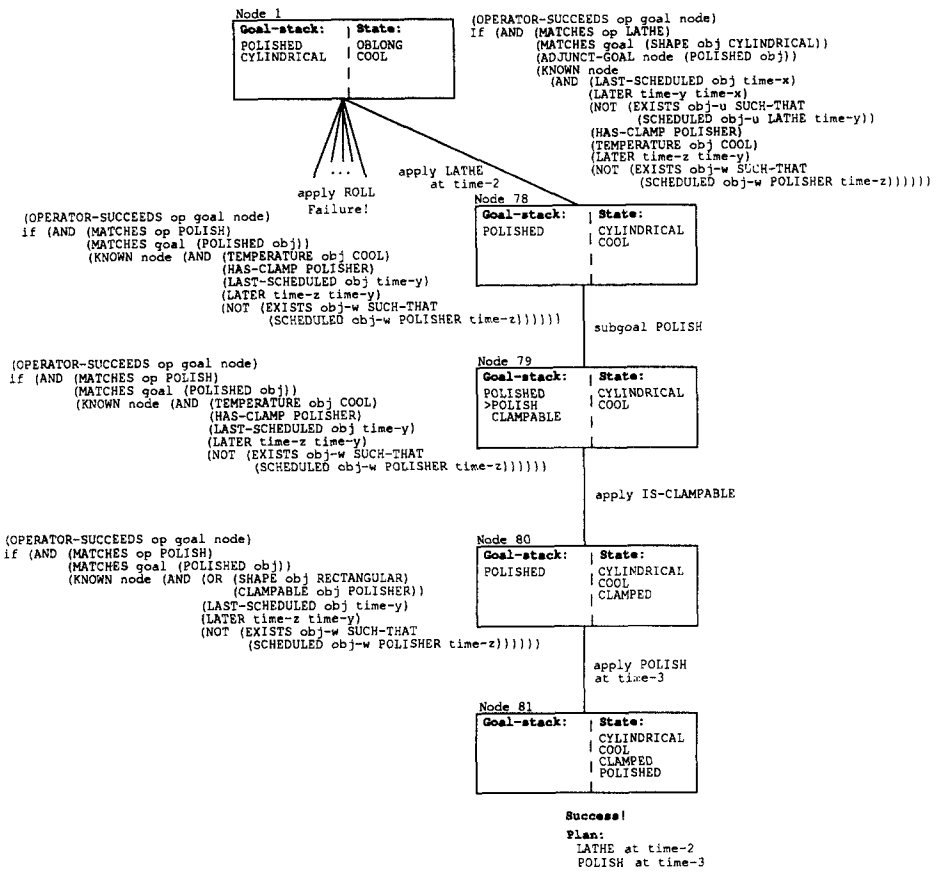

Fig. 15. Successful section of the search tree.

branch of the search tree). As we described in the last section, the only interesting example of success occurs at Node 1, where the problem solver chooses LATHE after having unsuccessfully tried ROLL. To explain LATHE's success, PRODIGY first explains why POLISH succeeded at Node 80 (as illustrated in the last section). Then this result is propagated up the tree as PRODIGY analyzes each node in turn. The result at each step of the analysis is shown next to each node in Fig. 15.

Let us briefly consider how the explanation process unfolds. The result from Node 80, which describes why POLISH was applicable at that node, is used as a lemma when explaining how the application of IS-CLAMPABLE at Node 79 enables the success of POLISH. The target concept and training example at Node 79 are shown below:

Target concept:
(OPERATOR-SUCCEEDS *op goal node*),

Training example:
(OPERATOR-SUCCEEDS POLISH (POLISHED OBJECT-A) Node 79)

As is evident from the problem solving trace, applying IS-CLAMPABLE at Node 79 was necessary to achieve the CLAMPABLE precondition of POLISHED. Therefore, when the system specializes OPERATOR-SUCCEEDS, Axiom-S2 (the recursive axiom for OPERATOR-SUCCEEDS) is retrieved:

(AND (APPLICABLE *next-op params node*)
(OPERATOR-SUCCEEDS *op goal child-node*)
(CHILD-NODE-AFTER-APPLYING-OP *child-node next-op node*))

First, APPLICABLE is specialized by a domain-level axiom specifying that the preconditions of IS-CLAMPABLE must be known to be true at *node*. Next, OPERATOR-SUCCEEDS is specialized by replacing it with the cached result from Node 80 with appropriate variable substitutions. Finally, specializing CHILD-NODE-AFTER-APPLYING-OP enables the simplifier to eliminate the CLAMPABLE precondition, since this was added by the application of IS-CLAMPABLE. (This last specialization accomplishes the "backpropagation" or "regression" of constraints that has been a familiar aspect of earlier EBL work.) The resulting expression, shown next to Node 79 in Fig. 15, states that POLISH will eventually succeed when the preconditions of the sequence IS-CLAMPABLE, POLISH are satisfied.

Because the result from Node 79 also holds at Node 78, no further explanation is necessary at Node 78. LATHE's success at Node 1 can now be explained. The analysis is similar to that described above, in effect, the

lower-level result from Node 78 is regressed across the LATHE operator. The final result states that LATHE is appropriate when the preconditions of the oeprator sequence LATHE, IS-CLAMPABLE, POLISH is applicable and the goal is to make an object cylindrical and then polish it. The control rule that is learned from this example is shown in Fig. 16. The rule is a preference rule, as are all rules learned from successes. (The analysis guarantees that the operator can solve the goals stated in the rule, not that it is the best operator under all circumstances.)

PRODIGY's method of learning from success is reminiscent of macro-operator formation.[10] In fact, variations of macro-operator learning have been employed by most other EBL problem solving systems to date. Unfortunately, we have found that the utility of this approach is fairly limited. Good performance depends on whether a small population of operator sequences can be identified that solve (or almost solve) most of the problems in the domain. Although PRODIGY's training example selection heuristics attempt to pick out only those training examples that illustrate particularly useful operator sequences (or subsequences), there may in fact be no small set of operator sequences that

```
IF (AND (CURRENT-NODE node)
        (CURRENT-GOAL node (SHAPE obj CYLINDRICAL))
        (CANDIDATE-OPERATOR node LATHE)
        (CANDIDATE-OPERATOR node other-op)
        (ADJUNCT-GOAL node (POLISHED obj))
        (KNOWN node
          (AND (LAST-SCHEDULED obj time-x)
               (LATER time-y time-x)
               (NOT (EXISTS obj-u SUCH-THAT
                      (SCHEDULED obj-u LATHE time-y)))
               (HAS-CLAMP POLISHER)
               (TEMPERATURE obj COOL)
               (LATER time-z time-y)
               (NOT (EXISTS obj-w SUCH-THAT
                      (SCHEDULED obj-w LATHE time-z))))))
THEN (PREFER OPERATOR LATHE OVER other-op)
```

Fig. 16. Preference rule learned by analyzing the success of LATHE.

[10] PRODIGY's method is not exactly macro-operator learning, since the learned rules only selected a single operator, not the entire operator sequence. In some cases macro-operators may be more efficient because the selection process happens once for the entire sequence. In other cases PRODIGY's control rules may be more efficient because they allow for more flexibility at each decision point, and because multiple control rules can be combined together by the simplifier. The various efficiency tradeoffs are discussed by Minton [52].

"covers" the domain. For this reason, we have included other methods of learning in PRODIGY, which distinguish it from most previous EBL systems. In the next section we show how learning from failure can be a valuable alternative to learning from success.

6.2. Learning from failure

In our machine-shop example, the problem solver constructed a plan to lathe and polish an object in order to make it cylindrical and polished. However, the problem solver did not immediately find this solution, but first explored the possibility of rolling the object and then polishing it. This possibility failed because the object could not be clamped to the polisher once it had been rolled due to its high temperature. In this section, we will describe how PRODIGY learns a control rule to avoid repeating this mistake in the future.

An illustrative subset of the axioms for explaining failures are shown in Fig. 17. (The axioms have been simplified for clarity.) They state that the failure of a node is implied by the failure of a goal at that node, which is in turn implied by failure of the available operators under all relevant bindings. The definition is recursive because bindings for an operator may fail if subgoaling generates a node that fails, or if applying the operator generates a node that fails. Other failure-related axioms (not shown in the figure) indicate that failure can occur if a loop is detected or a control rule rejects a candidate node, goal, operator or bindings. The full set of axioms for describing failures take up several pages, and are given in [52].

Figure 18 illustrates the portion of the failed search tree analyzed by PRODIGY as it explains the failure of ROLL at Node 1. The figure also shows for each of the nodes below Node 1 the intermediate results that describe why these nodes failed. As we have seen, PRODIGY begins the learning process by starting at the bottom left-most portion of the search tree and working upwards. The lower-level results serve as lemmas in explaining the higher-level failures.

As shown in Fig. 18, PRODIGY finds that Node 4 failed because there is no way to achieve the goal of cooling an object. (Objects can be heated as a side-effect of the ROLL operator.) This failure is propagated up the tree to Node 3, at which point PRODIGY can state that CLAMPING an object will fail if the object is not cool. Continuing the postorder traversal of the failed subtree, PRODIGY concludes that Node 5 failed because it is impossible to make an object rectangular. Thus Node 2 failed because polishing an object is impossible if the object is not rectangular and not cool (i.e., it is not stable and cannot be clamped). Finally, the top-level failure of ROLL is attributed to the fact the object had to be subsequently polished; as evidenced by the example, after applying ROLL, polishing is impossible. (Appendix C elaborates on PRODIGY's analysis of this example.)

Axiom-F1: A node fails if one of the goals at that node fails.
(NODE-FAILS *node*)
if (AND (ATOMIC-FORMULA *goal*)
(IS-GOAL *node goal*)
(GOAL-FAILS *goal node*))

Axiom-F2: A goal fails if all relevant operators fail to achieve it.
(GOAL-FAILS *goal node*)
if (FORALL *op* SUCH-THAT (IS-OPERATOR *op*)
(OPERATOR-FAILS *op goal node*))

Axiom-F3: An operator fails if it is irrelevant to the goal.
(OPERATOR-FAILS *op goal node*)
if (FORALL *effect* SUCH-THAT (IS-EFFECT *effect op params*)
(DOES-NOT-MATCH *effect goal* NIL))

Axiom-F4: An operator fails if it is relevant, but all bindings fail.
(OPERATOR-FAILS *op goal node*)
if (FORALL *bindings* SUCH-THAT
(IS-RELEVANT-BINDINGS *bindings op goal*)
(BINDINGS-FAIL *bindings op goal*))

Axiom-F5: A set of bindings fail if the operator cannot be applied with those bindings, and subgoaling fails.
(BINDINGS-FAIL *bindings op goal node*)
if (AND (NOT-APPLICABLE *bindings op node*)
(IS-CHILD-NODE-AFTER-SUBGOALING *child-node bindings op node*)
(NODE-FAILS *child-node*))

Axiom-F6: A set of bindings fail if applying the instantiated operator leads to failure.
(BINDINGS-FAIL *bindings op goal node*)
if (AND (IS-APPLICABLE *bindings op node*)
(IS-CHILD-NODE-AFTER-APPLYING-OP *child-node bindings op node*)
(NODE-FAILS *child-node*))

Fig. 17. Illustrative architecture-level axioms for FAILS.

The top-level result, which describes why applying ROLL at Node 1 failed, is converted directly into an operator rejection rule via the rule construction template for OPERATOR-FAILS. The resulting control rule was shown earlier in Fig. 8. (In addition, all of the node failure descriptions in Fig. 18 are converted into node rejection control rules, although our top-level result is by far the

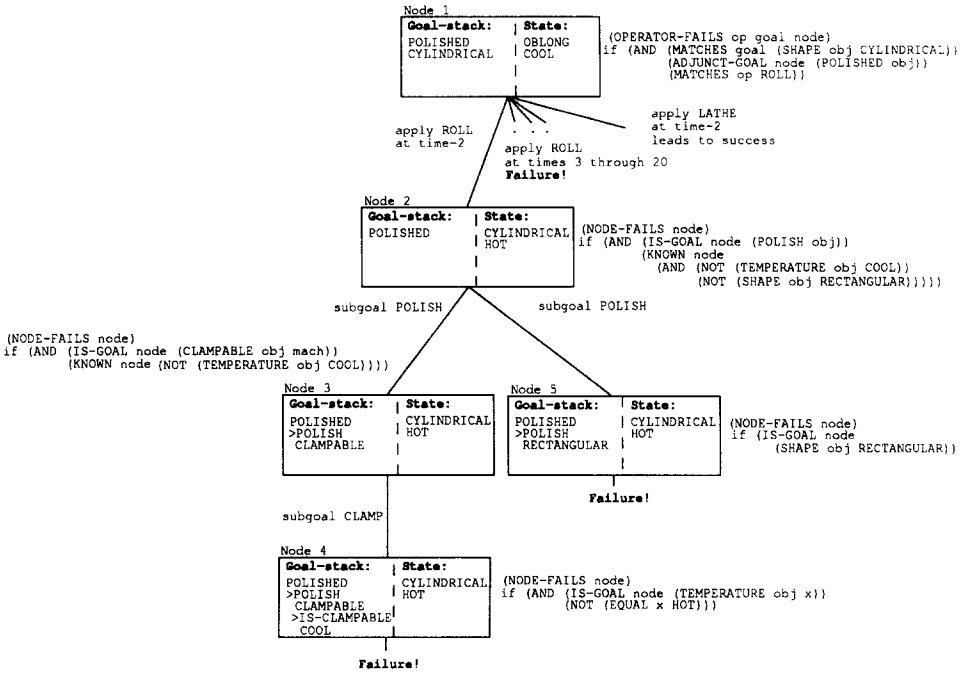

Fig. 18. Failed section of the search tree.

most useful.) This rule has high utility because its match cost is low, it is frequently applicable, and there is a large payoff when the rule is applicable. Specifically, by eliminating the need to consider the ROLL operator, the rule produces a savings on the order of $N \times T$, where N is the size of the schedule, and T is the cost of considering ROLL for a particular time period (the cost of searching Nodes 2 through 5 in Fig. 18).

In general, failure-driven learning is most beneficial when the reasons for the failure can be expressed by a concise, easily evaluated description. Notice that the rule learned from this example provides a much more general and efficient means for choosing between ROLL and LATHE than was gained from analyzing LATHE's success.

Our example also demonstrates that when a failure occurs, it may not be necessary to analyze the entire failed subtree to learn from the failure. To learn why applying ROLL at Node 1 failed, it was only necessary to analyze the subtree rooted at Node 2. Once this was accomplished PRODIGY immediately found that the other applications of ROLL (at TIME-2 through TIME-20) failed for the same reason. Thus, it was not necessary to construct separate explanations for each of these failures. In fact, had learning been interleaved with problem solving PRODIGY could have avoided these subsequent attempts to apply ROLL, thanks to the control rule learned from the failure of Node 2.

Thus, when learning and problem solving are interleaved, PRODIGY effectively carries out a general form of dependency-directed backtracking [24].

6.3. Learning from goal interference

Goal interactions are ubiquitous in planning. Goals may either interact constructively, or they may interfere with each other. This latter case can create serious difficulties for a domain-independent planner. For this reason, many previous planning systems have included built-in heuristics for avoiding and/or recovering from interferences [9, 70, 85]; PRODIGY improves on this by reasoning about interferences in order to learn domain-specific control rules. This type of learning method was pioneered by Sussman's HACKER program [79].

In order to illustrate how goals can interfere, we will modify our scheduling example. Let us assume that PRODIGY attempted to solve the goal (POLISHED OBJECT-A) before the goal (SHAPE OBJECT-A CYLINDRICAL), as shown in Fig. 19. (This figure explicitly shows these top-level goals as a conjunction, which is intended to illustrate that they are initially unordered and then PRODIGY selects a goal to solve first. Our previous figures did not show the top-level conjunction.) Thus, after successfully polishing the object, the system will attempt to reshape it. However, both lathing the object and rolling the object will have the unfortunate side-effect of deleting (POLISH OBJECT-A). This is referred to as a *goal protection violation*. Of course, PRODIGY can re-polish the object, in which case the resulting plan will involve polishing the object twice. However, because the EBL module notices the protection violation that occurs

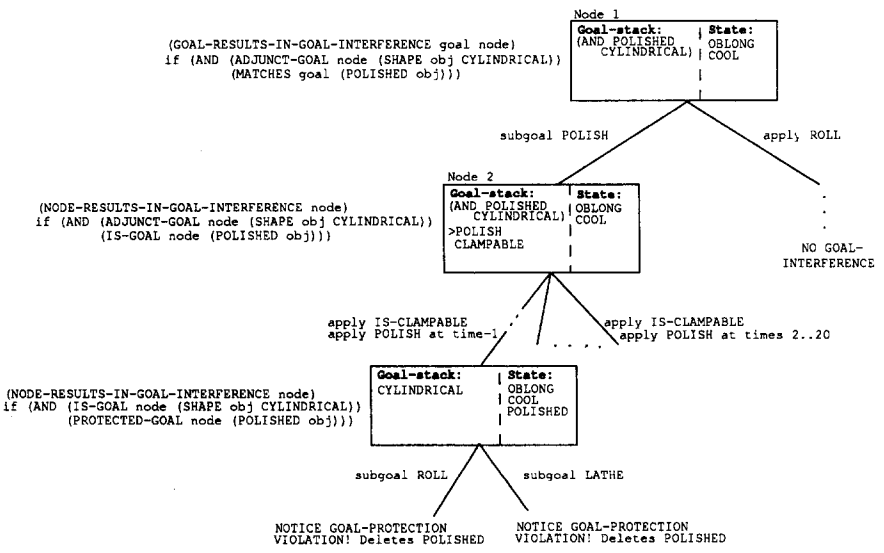

Fig. 19. Section of search tree illustrating goal interference.

when (POLISHED OBJECT-A) is deleted, PRODIGY can learn to order the goals correctly.

In the general case, we say that a plan exhibits goal interference if there is a goal in the plan that has been negated by a previous step in the plan. There are two complementary forms in which goal interference may manifest itself during the planning process. As we have seen, a *protection violation* occurs when an action undoes a previously achieved goal, requiring the goal to be re-achieved. A *prerequisite violation*, on the other hand, occurs when an action negates a goal that arises later in the planning process. Goal interferences may result in suboptimal plans or the outright failure to find a plan. In this section we will only consider the first case, since learning from failure is covered by the axioms described the previous section.

Since goal interferences can be unavoidable in some problems (and may even occur in optimal plans) PRODIGY learns rules that express preferences, thereby enabling solutions to be found even when interferences are unavoidable. When goal interferences can be avoided search time is typically reduced, and in addition, better solutions tend to be found. Learning to avoid goal interference is an optimization technique specifically designed for planning domains; in other types of domains (e.g., theorem proving) this technique may be irrelevant because interferences may not occur.

Returning to our example, to explain why the initial goal ordering resulted in goal interference, PRODIGY must show that all paths in the search tree subsequent to that goal selection decision resulted in either a failure or a goal interaction.[11] To formulate its explanation, PRODIGY will use the target concept GOAL-RESULTS-IN-GOAL-INTERFERENCE, which describes why selecting a goal at a node results in goal interference. The tree shown in Fig. 19 is annotated with some of the intermediate results derived during the EBS process. The high-level axioms for explaining goal interferences are shown in Appendix D. The axioms that are used to explain goal interferences are similar to (and overlap with) the axioms for failure, with the addition of several low-level axioms describing how protection and prerequisite violations occur.

As shown by Fig. 19, PRODIGY first notices that both ROLL and LATHE clobber the POLISHED property, causing a goal protection violation. In other words, all operators for making an object cylindrical result in a goal interfer-

[11] In normal problem solving mode, the PRODIGY problem solver will be quite content with a suboptimal solution that exhibits a goal interference, and will not continue searching for an optimal solution. Therefore, during the learning phase, if the solution exhibits a goal interference, the EBL module will call upon the problem solver to resume its search until all paths below the suspect decision point exhibit a protection violation, prerequisite violation or failure (i.e., until the example of goal interference has been verified). This is accomplished by the *training example recognition function* for goal interference. (Each target concept is associated with such a function, which is the code that actually picks out the training examples for that concept at a node.) The process of recognizing goal interferences and selecting appropriate training examples is discussed in more depth by Minton [52].

ence once the POLISHED property of the object has been achieved. Thus, at Node 2, the interaction is attributed to the fact that (SHAPE *obj* CYLINDRICAL) was an adjunct goal, and therefore slated to be achieved subsequently to (POLISHED *obj*). (The alternative paths emanating from Node 2 are not analyzed by the learning component because the system immediately determines that the reasons for the observed interaction are valid for each of them as well.) Finally at Node 1, PRODIGY determines that selecting a goal matching (POLISHED *obj*) will result in an interaction if (SHAPE *obj* CYLINDRICAL) must subsequently be achieved. The control rule resulting from the example is shown below, and states that PRODIGY should prefer the goal (SHAPE *obj* CYLINDRICAL) to the goal (POLISHED *obj*) when it is considering which goal to work on first.

```
IF (AND (CURRENT-NODE node)
        (CANDIDATE-GOAL node (SHAPE obj CYLINDRICAL))
        (CANDIDATE-GOAL node (POLISHED obj)))
THEN (PREFER GOAL (SHAPE obj CYLINDRICAL) OVER
        (POLISHED obj))
```

7. Comparison with Related Work

While the PRODIGY system illustrates one method of integrating EBL and problem solving, it is useful to analyze some other learning systems to compare how they exploit EBL. We will consider four prominent approaches, and compare how they address each of the issues outlined in Section 3: target concept coverage, scope of the theory, method of constructing explanations from examples, and the operationality criterion.

Perhaps the most widely known work on EBL is the EBG method by Mitchell et al. [56]. The main thrust of their effort was to clarify and unify much of the earlier research on explanation-based learning. As such, EBG is mainly a framework that is not closely tied to a particular performance system. However, there have been several implementations of EBG that do address the issues relevant to combining problem solving and learning. An alternate proposal for EBL, presented by DeJong and Mooney [20], discusses some deficiencies of and alternatives to Mitchell et al.'s EBG method. Their approach is primarily oriented towards schema-based problem solving. Among other points, they emphasize the necessity for a realistic operationality criterion appropriate to their problem solving architecture. They also describe techniques for improving explanations that are originally produced by observing the solution to a problem.

A third approach is illustrated by the SOAR system, developed by Laird, Newell, and Rosenbloom [42]. SOAR was not developed explicitly as an EBL system; instead, it is intended to be a general cognitive architecture utilizing an independently developed learning mechanism called chunking. Chunking oper-

ates by summarizing the information examined while processing each subgoal. The mechanism bears a strong resemblance to EBL, and Rosenbloom and Laird [69] have described how EBL can be implemented via chunking in SOAR. Finally, a discussion of EBL problem solving systems could not be complete without mentioning STRIPS [29]. Even though STRIPS far preceded the recent surge of work on EBL, the MACROP learning technique is now recognized as a simple form of EBL. In comparing these EBL approaches, we will only investigate how they have been integrated with a problem solver in order to optimize problem solving performance. However, we note that all of these systems address additional issues not considered here.

7.1. Target concepts

First, let us consider each system's target concepts to evaluate the diversity of the optimization strategies that have been pursued. Recall that a system's target concepts reflect the range of situations that can be optimized. Most of the implemented systems have demonstrated only a single strategy, namely learning from success. This includes STRIPS, LEX2 [54, 56] (an early implementation of EBG), and the systems discussed by DeJong and Mooney (e.g., GENESIS). These systems equate their target concepts with problem solving goals. They therefore learn descriptions of the states for which a specified operator (or sequence thereof) leads to success. Only recently have other optimization strategies begun to be considered. For example, Mostow and Bhatnagar [62] use EBG to learn from problem solving failures in their FAILSAFE system. In a related vein, DeJong and Mooney have also proposed using EBL to refine over-general concepts (as opposed to merely operationalizing them), which goes beyond the standard notion of learning from success. This proposal has been recently extended and implemented by Chien [14]. Even so, each of these systems employ a single optimization strategy, as opposed to PRODIGY, which employs multiple optimization strategies. PRODIGY currently learns from successes, failures, and goal interactions. Also, the set of target concepts are declaratively specified and thus easily extensible.

The one system, in addition to PRODIGY, that may be regarded as using multiple optimization strategies is SOAR. On one level, SOAR uses a single optimization technique, chunking. Chunking operates on the production level; whenever a result is returned for a goal, SOAR records the conditions matched by the productions that produced the result. Thus chunking is essentially a form of caching. However, if one looks at SOAR at a higher level of abstraction, chunking can be regarded as implementing alternate optimization strategies. For example, the system is able to learn from certain types of failures as it considers how to solve a goal. In particular, SOAR can learn to avoid an operator that leads to failure so long as there is a specific reason for the failure (returned as the result). Thus, explicit failure in the base domain

can give rise to a preference that the choice is undesirable, and a chunk may be learned summarizing that conclusion. SOAR cannot learn if the failure is caused by exhausting all available alternatives [78]. Also, Rosenbloom, Laird and Newell have not addressed the use of explicit optimization goals, which would require the system to have a theory (i.e., problem space) describing itself.

7.2. Theory

The theory used by an EBL problem solving system provides a means for proving that an optimization applies in a given problem solving episode. There is a spectrum of approaches to providing a theory to drive the EBL process. One end of the spectrum is best exemplified by STRIPS, whose theory is completely represented by its domain operators. The system's "proofs" are not proofs in the usual sense, but sequences of domain operators.[12] The limited scope of the theory reflects the fact that STRIPS' target concepts are limited to the goals of the problem solver. At the other end, PRODIGY uses an explicit theory describing the relevant aspects of the problem solver in addition to a theory of the domain. This is necessary because PRODIGY's target concepts are meta-level problem solving phenomena (i.e., problem solving failures, goal interactions, etc.), rather than simply base-level problem solving goals. Since PRODIGY's theory is declarative and external to the problem solver, it can be extended easily to accommodate new target concepts. By contrast, STRIPS' domain operators and learning component are intimately intertwined with the problem solving system.

The other EBL systems can also be classified within this spectrum. The EBG approach clearly involves a separate, fully declarative theory describing the performance system, as illustrated by the LEX2 and FAILSAFE implementations. By contrast, the core of SOAR (i.e., the matcher, decision procedure, etc.) is not encoded as a declarative theory or problem space, it is only expressed in terms of the set of productions that implement the system. Thus, a theory of the problem solver is not used in the EBL process, and so there are some aspects of itself that SOAR cannot improve. This may be attributed to the view of chunking as a pervasive and automatic compilation mechanism that should not reflect on the workings of the agent. Laird, Rosenbloom and Newell [42] thus characterize SOAR as a "simple experience learner" rather than a "deliberate learner."[13] Finally, in DeJong and Mooney's scheme, as in STRIPS, explanations are constructed by observing operator sequences, and consequently the theory is limited to the task domain. To a certain extent, DeJong and Mooney expand on the STRIPS approach, because explanations may also

[12] As DeJong and Mooney [20] point out, however, operator sequences and proofs are isomorphic.

[13] It might be reasonable in SOAR to provide problems spaces for reasoning about itself. This would allow chunking to implement a wider variety of optimization strategies.

include inference rules and related information describing relevant features of the task domain. However, as with STRIPS, this approach operates by analyzing why a plan achieved a goal; thus the theory does not describe the problem solver itself, or allow learning based on other target concepts.

7.3. Mapping from an example to an explanation

One of the significant practical requirements for any EBL system is that it be able to efficiently construct explanations from examples. PRODIGY uses the search tree generated by the problem solver to control the construction of the explanation. This mapping is done efficiently by the EBS method's discriminator functions. In contrast, the EBG method does not specify any particular mechanism for identifying or constructing explanations. Recent implementations of EBG [36, 62] have relied on a theorem prover to construct explanations. However, if the theorem prover ignores the experience gained during problem solving, it must re-explore the search space traversed by the problem solver. In fact, as mentioned by Mostow and Bhatnagar [62], when learning from failure the explanation facility could discover a reason for an example's failure that is different from the one encountered by the problem solver.

For STRIPS and SOAR, the cost of mapping an example to an explanation is small. This is because the explanation is built directly from the sequence of operators or productions formed by the problem solver in a problem solving episode. That is, the explanation-based learner simply uses the results of the search already done by the problem solver. Similarly, in DeJong and Mooney's EBL scheme (as described in [20]), the explanation process starts with an observed operator sequence. However, the operator sequence is optional; if it is not available, an explanation will be built from scratch, as in EBG. Furthermore, the initial explanation can be improved so that a more operational and/or general schema is learned. (Recent extensions to their method for improving explanations are also described by Shavlik [74] and Mooney [60].) DeJong and Mooney do not discuss the costs involved in the explanation construction process in detail. However, their strategy of relying on observation whenever possible reduces the expense of the explanation construction process.

7.4. Operationality

A system's operationality criterion must reflect the computational costs and benefits of the control knowledge that is learned. However, the issue has been largely ignored in explanation-based learning. It is often assumed that the knowledge learned by an EBL problem solving system will improve the system's performance, but this may be an optimistic oversimplification. In general, the optimization techniques that have been implemented by EBL are *heuristic strategies*, not guaranteed to produce improvement in all circum-

stances. For example, the STRIPS MACROPS learning technique can actually decrease efficiency; if the cumulative time cost of testing the macro-operators preconditions outweighs the benefits in reduced search, then overall performance can decrease [49].

Mitchell et al.'s EBG proposal attempts to deal with this problem by including an explicit operationality criterion for testing an explanation's utility. However, as pointed out by DeJong and Mooney, the type of operationality criterion used by Mitchell et al. assures only that the new knowledge can be directly evaluated, it does not guarantee that it is actually useful. In the EBG examples, the operationality criterion simply specified that the explanation must refer only to directly computable or observable features. Later work on META-LEX extends the operationality criterion by considering the context of the learning [39]. In DeJong and Mooney's scheme, the cost of testing a "feature" may vary depending on the system's knowledge. Thus, their operationality criterion is dynamic, rather than static. PRODIGY refines the notion of operationality even further, specifying that the learned information must improve the efficiency of the problem solver. PRODIGY uses an explicit utility metric, described in Section 5.4, that depends on the time cost of evaluating, or matching, control knowledge compared to the time savings produced by that knowledge.

A very different approach to the operationality issue has been taken by the designers of SOAR, who, for the most part, intentionally ignore the utility issue. In part, this is because chunking is presumed to be an automatic process, and in part, because performance in SOAR is measured by the number of decisions necessary to perform a task. Since chunking will reduce the number of decisions that are made (so long as there is any overlap between tasks), it does very well according to its performance metric. However, making a decision may involve complex processing in its own right. For example, the cost of matching chunks is ignored in this simplistic metric; therefore, adding arbitrarily many and arbitrarily complex chunks can never hurt performance according to this metric. For this reason, the number of decisions may not correlate with actual CPU time, at least on conventional machines. The relationship between CPU time and decisions is a complex issue that depends in part upon the assumptions about the type of chunks typically formed. This issue has recently been investigated by Tambe and Newell [81], who have shown that in certain task domains SOAR will indeed learn expensive chunks that prolong the time per decision. Based on this work, Tambe and Rosenbloom [82] have considered restricting the language in which chunks are expressed to alleviate this problem.

8. Concluding Remarks

Having presented an in-depth investigation of explanation-based learning and its application to improving problem solving performance, let us return to a broader perspective. In essence, EBL enables the learning mechanism to take

advantage of domain knowledge that specifies *why* some example is an instance of a target concept. This knowledge may not be available in all applications, but when it is available, it can serve as a strong means of guiding the generalization process. If we look at inductive learning, an alternative paradigm for learning from examples, the role of domain knowledge is much less direct. Inductive learning programs compare examples of a target concept in order to find a description that is consistent with all the examples, regardless of whether the common structure is coincidental or causally mandated. Thus, in inductive learning, the role of domain knowledge is merely to delimit the descriptions that the program considers, or at best to provide a heuristic bias for preferring certain descriptions over others among several consistent with the training data. For this reason, EBL is generally a preferable method if there is a strong source of domain knowledge available to guide the learning process.

One reason that EBL is an appropriate method for improving problem solving performance is that the necessary knowledge can be made available to the learner. In operator-based problem solving architectures, such as PRODIGY, the knowledge is provided by axiomatizing the domain operators and crucial aspects of the problem solving architecture itself. As we have shown in the paper, PRODIGY can use this theory to synthesize effective search control rules with the aid of problem solving traces. Since the descriptions of the task operators can be converted automatically into logic, the main requirement for incorporating EBL into a problem solver is to decide on appropriate target concepts, and then axiomatize the relevant aspects of the problem solving architecture, as we have done with PRODIGY.

We envision many extensions to our present EBL results, including scaling up to much larger task domains and learning from a partial domain theory (e.g., one where not all consequences or preconditions of an operator are known with certainty). Of particular interest is the integration of EBL with related parallel efforts that share the same PRODIGY problem solver: learning by derivational analogy [11] and learning to formulate effective abstractions for multi-level planning [41]. Both EBL and analogy strive to exploit past experience in order to reduce future search; therefore it is unclear whether a combination of the two will prove more effective than either in isolation. Plan abstraction, on the other hand, exploits orthogonal knowledge sources and therefore may yield complementary savings in search time, if EBL is applied to learn control rules at each level of abstraction. In all these extensions, however, we expect the underlying explanation-based learning mechanism to remain the same, allowing knowledge to play a strong role in the learning process.

Appendix A. The PRODIGY Architecture

PRODIGY is a general problem solver combined with several learning modules. The PRODIGY architecture, in fact, was designed both as a unified testbed for

different learning methods—including the EBL module discussed in this paper—and as a general architecture to solve interesting problems in complex task domains. Let us now focus on the architecture itself, as diagrammed in Fig. 20.

The operator-based problem solver produces a complete search tree, encapsulating all decisions—right ones and wrong ones—as well as the final solution. This information is used by each learning component in different ways, including the EBL component, which learns search control rules. In addition to the central problem solver,[14] PRODIGY has the following learning components:

- A user interface that can participate in an apprentice-like dialogue, enabling the user to evaluate and guide the system's problem solving and learning. The interface is graphic-based and tied directly to the problem solver, so that it can accept advice as it is solving a problem (i.e., coaching) or replay and analyze earlier solution attempts, all-the-while refining the factual or control knowledge.
- An explanation-based learning facility [52] for acquiring control rules from a problem solving trace, as indicated in Fig. 20. Explanations are constructed from an axiomatized theory describing both the domain and relevant aspects of the problem solver's architecture. Then the resulting descriptions are expressed

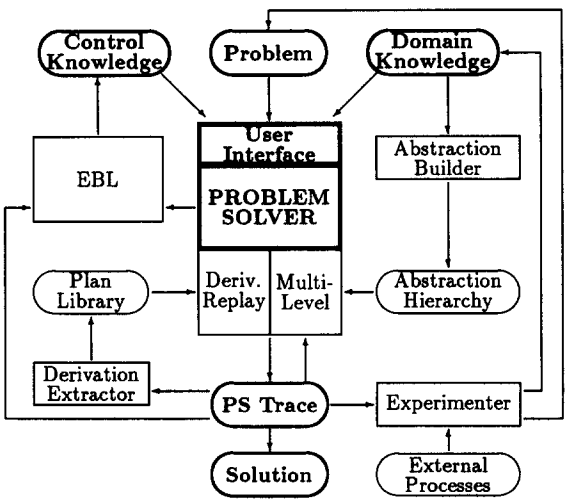

Fig. 20. The PRODIGY architecture: multiple learning modules unified by a common representation language and a shared general problem solver.

[14] The problem solver is an advanced operator-based planner that includes a simple reason-maintenance system and allows operators to have conditional effects. The problem solver's search (means-ends analysis) is guided by explicit domain-independent and domain-specific control rules. All of PRODIGY's learning modules share the same general problem solver and the same knowledge representation language, PDL.

in control rule form, and control rules whose utility in search reduction outweighs their application overhead are retained. The EBL method is discussed at length in this article.

– A derivational analogy engine [11, 23] that is able to replay entire solutions to similar past problems, calling the problem solver recursively to reduce any new subgoals brought about by known differences between the old and new problems. As indicated in Fig. 20, analogy and EBL are independent mechanisms to acquire domain-specific control knowledge. They coexist in PRODIGY and should be more tightly coupled than in the present architecture.

– A multi-level abstraction planning capability [41]. First, the axiomatized domain knowledge is divided into multiple abstraction levels based on an in-depth analysis of the domain. Then, during problem solving, PRODIGY proceeds to build abstract solutions and refine them by adding back details from the domain, solving new subgoals as they arise. This method is orthogonal to analogy and EBL, in that both can apply at each level of abstraction.

– A learning-by-experimentation module for refining domain knowledge that is incompletely or incorrectly specified [12]. Experimentation is triggered when plan execution monitoring detects a divergence between internal expectations and external expectations. As indicated in the figure, the main focus of experimentation is to refine the factual domain knowledge, rather than the control knowledge.

The problem solver and EBL component of PRODIGY have been fully implemented, and tested on several task domains including the blocks world domain, a machine-shop scheduling domain, and a 3-D robotics construction domain. The other components, while successfully prototyped, are at various stages of development and implementation.

Appendix B. Partial Specification of the Machine-Shop Scheduling Domain

This appendix contains the definitions of the operators and inference rules that were used in the examples in this paper. A complete specification of all the operators and inferences rules for this domain can be found in [52].

Example operators

```
;Lathing makes an object cylindrical.

(LATHE (obj time)
  (PRECONDITIONS
    (AND
      (LAST-SCHEDULED obj prev-time)
      (LATER time prev-time)
      (NOT (EXISTS other-obj SUCH-THAT
             (SCHEDULED other-obj LATHE time)))))
```

```
    (EFFECTS
      (DELETE (SHAPE obj old-shape))
      (DELETE (LAST-SCHEDULED obj prev-time))
      (IF (POLISHED obj)
          (DELETE (POLISHED obj)))
      (ADD (LAST-SCHEDULED obj time))
      (ADD (SHAPE obj CYLINDRICAL))
      (ADD (SCHEDULED obj LATHE time))))

;Rolling makes an object cylindrical, and heats it

(ROLL (obj time)
  (PRECONDITIONS
    (AND
      (LAST-SCHEDULED obj prev-time)
      (LATER time prev-time)
      (NOT (EXISTS other-obj SUCH-THAT
              (SCHEDULED other-obj ROLLER time)))))
  (EFFECTS
    (DELETE (TEMPERATURE obj old-temp))
    (DELETE (SHAPE obj old-shape))
    (DELETE (LAST-SCHEDULED obj prev-time))
    (IF (POLISHED obj)
        (DELETE (POLISHED obj)))
    (ADD (TEMPERATURE obj HOT))
    (ADD (LAST-SCHEDULED obj time))
    (ADD (SHAPE obj CYLINDRICAL))
    (ADD (SCHEDULED obj ROLLER time))))

;Before polishing, nonrectangular objects must be clamped to the polisher.

(POLISH (obj time)
  (PRECONDITIONS
    (AND
      (OR (SHAPE obj RECTANGULAR)
          (CLAMPABLE obj POLISHER))
      (LAST-SCHEDULED obj prev-time)
      (LATER time prev-time)
      (NOT (EXISTS other-obj SUCH-THAT
              (SCHEDULED other-obj POLISHER time)))))
  (EFFECTS
    (DELETE (LAST-SCHEDULED obj prev-time))
    (ADD (LAST-SCHEDULED obj time))
    (ADD (POLISHED obj))
    (ADD (SCHEDULED obj POLISHER time))))
```

An example inference rule

```
;An object can be clamped to a machine only if the object is cool and the
machine has a clamp.

(IS-CLAMPABLE (obj machine)
  (PRECONDITIONS
    (AND (TEMPERATURE obj COOL)
         (HAS-CLAMP machine)))
  (EFFECTS
    (ADD (CLAMPABLE obj machine))))
```

Appendix C. Explaining Failures: An In-Depth Example

In this appendix our intention is to provide the reader with a detailed illustration of PRODIGY's explanation process on an interesting example. For this purpose, we describe in more depth how PRODIGY explains the failure of Node 4 in our second example, originally presented in Section 6.2. As we have stated, Node 4 failed because the goal was (TEMPERATURE OBJECT-A COOL), and no operator is available for cooling objects. In fact, the only operator that changes an object's temperature is ROLL, which heats the object. Figure 21 lists the relevant lower-level architectural axioms needed to explain why the operators' postconditions did not match the goal. Normally, as we will see, much of this detail simplifies out during the specialization process.

The EBS process begins with the following target concept and example:

```
Target concept: (NODE-FAILS node)

Example: (NODE-FAILS Node 4)
```

NODE-FAILS is specialized by architectural-level Axiom-F1 and Axiom-F2 (described earlier in Fig. 17), to arrive at the following:

```
(NODE-FAILS node)
if (AND (ATOMIC-FORMULA goal)
        (IS-GOAL node goal)
        (FORALL op SUCH-THAT (IS-OPERATOR op)
          (OPERATOR-FAILS op goal node)))
```

After specializing ATOMIC-FORMULA and simplifying IS-OPERATOR, we have the expression shown below. The description of the goal (TEMPERATURE *obj temp*), comes from the specialization of ATOMIC-FORMULA. Each of the references to OPERATOR-FAILS come from the simplifier's expansion of IS-OPERATOR.

The following axioms are used to explain why a match failed. The arguments to the matcher are a pattern (an atomic formula with variables), such as (TEMPERATURE *y x*), a list that is a ground formula, such as (TEMPERATURE OBJECT-A COOL), and an initial set of bindings for the variables in the pattern. A match fails if the list cannot be unified with the pattern.

Axiom MATCH-FAIL-1: A match fails if the head element of the pattern is a constant that is not equal to the head element of the list.
(DOES-NOT-MATCH *pattern list bindings*)
if (AND (HEAD-ELEMENT *list-head list*)
(HEAD-ELEMENT *pat-head pattern*)
(IS-CONSTANT *pat-head*)
(NOT (EQUAL *list-head pat-head*)))

Axiom MATCH-FAIL-2: A match fails if the head element of the pattern is a variable, and its binding is not equal to the head element of the list.
(DOES-NOT-MATCH *pattern list bindings*)*C*
if AND (HEAD-ELEMENT *list-head list*)
(HEAD-ELEMENT *pat-head pattern*)
(IS-VARIABLE *pat-head*)
(IS-BINDING *value pat-head bindings*)
(NOT (EQUAL *value list-head*)))

Axiom MATCH-FAIL-3: A match fails if the head element of the pattern matches the head element of the list, but the remainder of the list does not match the remainder of the pattern.
(DOES-NOT-MATCH *pattern list bindings*)
if (AND (HEAD-ELEMENT *list-head list*)
(HEAD-ELEMENT *pat-head pattern*)
(TAIL *rest-of-list list*)
(TAIL *rest-of-pattern pattern*)
(UPDATE-BINDINGS *new-bindings list-head pat-head bindings*)
(DOES-NOT-MATCH *rest-of-pattern rest-of-list new bindings*))

Fig. 21. Architecture-level axioms for DOES-NOT-MATCH.

(NODE-FAILS *node*)
if (AND (IS-GOAL *node* (TEMPERATURE *obj temp*))
(OPERATOR-FAILS LATHE (TEMPERATURE *obj temp*) *node*)
(OPERATOR-FAILS POLISH (TEMPERATURE *obj temp*) *node*)
(OPERATOR-FAILS IS-CLAMPABLE (TEMPERATURE *obj temp*) *node*)
(OPERATOR-FAILS ROLL (TEMPERATURE *obj temp*) *node*))

Except for ROLL, no operator or inference rule has a postcondition that can match (TEMPERATURE *obj temp*). Therefore after specialization, each of these operator failures (other than that of ROLL) simplify to TRUE, and we are left with the intermediate result shown below. (We omit a detailed discussion of this process in order to focus on the remainder of the example.)

(NODE-FAILS *node*)
if (AND (IS-GOAL *node* (TEMPERATURE *obj temp*))
(OPERATOR-FAILS ROLL (TEMPERATURE *obj temp*) *node*))

OPERATOR-FAILS is then specialized by Axiom-F3 (see Fig. 17), which states that an operator fails if its postconditions (i.e., effects) do not match the goal:

(NODE-FAILS *node*)
if (AND (IS-GOAL *node* (TEMPERATURE *obj temp*))
(FORALL *effect* SUCH-THAT (IS-EFFECT *effect op params*)
(DOES-NOT-MATCH *effect goal* NIL)))

Then the simplifier expands IS-EFFECT, giving the following:

(NODE-FAILS *node*)
if (AND (IS-GOAL *node* (TEMPERATURE *obj temp*))
(DOES-NOT-MATCH (TEMPERATURE *obj* HOT)
(TEMPERATURE *obj* temp) NIL)
(DOES-NOT-MATCH (LAST-SCHEDULED *obj time*)
(TEMPERATURE *obj temp*) NIL)
(DOES-NOT-MATCH (SHAPE *obj* CYLINDRICAL)
(TEMPERATURE *obj temp*) NIL)
(DOES-NOT-MATCH (SCHEDULED *obj* ROLLER *time*)
(TEMPERATURE *obj temp*) NIL))

The first of the DOES-NOT-MATCH terms is successively specialized by Axiom MATCH-FAIL-3 (two applications) and MATCH-FAIL-2, because this match failed because COOL was not equal to HOT. The remaining DOES-NOT-MATCH terms are specialized by MATCH-FAIL-1, because the predicates are not equal. After simplifying we have the following final result:

(NODE-FAILS *node*)
if (AND (IS-GOAL *node* (TEMPERATURE *obj temp*))
(NOT (EQUAL HOT *temp*)))

Appendix D. Representative Axioms for Explaining Goal Interference

The following axioms illustrate the recursive nature of the GOAL-INTERFERENCE analysis.

Axiom-I1: A node results in goal interference if there exists a set of goals at that node that mutually interfere.
(NODE-RESULTS-IN-GOAL-INTERFERENCE *node*)
if (AND (SET-OF-GOALS *goals node*)
(FORALL *goal* SUCH-THAT (MEMBER *goal goals*)
(AND (GOAL-RESULTS-IN-GOAL-INTERFERENCE *goal node*)
(INTERFERENCE-DEPENDS-ON-GOAL-SET *goal node goals*))))

Axiom-I2: A goal results in goal interference if all operators result in goal interference once that goal is selected.
(GOAL-RESULTS-IN-GOAL-INTERFERENCE *goal node*)
if (FORALL *op* SUCH-THAT (IS-OPERATOR *op*)
(OPERATOR-RESULTS-IN-GOAL-INTERFERENCE *op goal node*))

Axiom-I3: An operator results in goal interference if all bindings for that operator result in goal interference.
(OPERATOR-RESULTS-IN-GOAL-INTERFERENCE *op goal node*)
if (FORALL *bindings* SUCH-THAT
(IS-RELEVANT-BINDINGS *bindings op goal*)
(BINDINGS-RESULT-IN-GOAL-INTERFERENCE *bindings op goal node*))

Axiom-I4: A set of bindings for an operator results in goal interference if the operator cannot be applied with those bindings, and subgoaling results in goal interference.
(BINDINGS-RESULT-IN-GOAL-INTERFERENCE *bindings op goal node*)
if (AND (NOT-APPLICABLE *bindings op node*)
(IS-CHILD-NODE-AFTER-SUBGOALING *child-node bindings op node*)
(NODE-RESULTS-IN-GOAL-INTERFERENCE *child- node*))

Axiom-I5: A set of bindings for an operator results in goal interference if applying the instantiated operator results in goal interference.
(BINDINGS-RESULT-IN-GOAL-INTERFERENCE *bindings op goal node*)
if (AND (IS-APPLICABLE *bindings op node*)
(IS-CHILD-NODE-AFTER-APPLYING-OP *child-node bindings op node*)
(NODE-RESULTS-IN-GOAL-INTERFERENCE *child-node*))

As described in Section 6.3, to prove that goal interference occurs, PRODIGY must show that all paths result in failure or interference. The next four axioms illustrate how the definition of goal interference refers to the axioms for failure.

Axiom-I6: A node results in goal interference if the node fails.
(NODE-RESULTS-IN-GOAL-INTERFERENCE *node*)
if (NODE-FAILS *node*)

Axiom-I7: A goal results in goal interference if the goal fails.
(GOAL-RESULTS-IN-GOAL-INTERFERENCE *goal node*)
if (GOAL-FAILS *goal node*)

Axiom-I8: An operator results in goal interference if the operator fails.
(OPERATOR-RESULTS-IN-GOAL-INTERFERENCE *op goal node*)
if (OPERATOR-FAILS *op goal node*)

Axiom-I9: A set of bindings results in goal interference if the bindings fail.
(BINDINGS-RESULTS-IN-GOAL-INTERFERENCE *bindings op goal node*)
if (BINDINGS-FAIL *bindings op goal node*)

ACKNOWLEDGMENT

The authors gratefully acknowledge the help of Michael Miller, Henrik Nordin, and Ellen Riloff in implementing the PRODIGY system and the contributions of the other members of the project, Robert Joseph, Alicia Perez, Santiago Rementeria, and Manuela Veloso. Comments and suggestions by Ranan Banerji, Murray Campbell, Jerry DeJong, Smadar Kedar-Cabelli, Rich Keller, Sridhar Mahadevan, Tom Mitchell, Jack Mostow, Allen Newell, David Steier, and Prasad Tadepalli have aided the development of the PRODIGY system, and the writing of this paper. Finally, we would like to thank the anonymous reviewers for their detailed comments on the paper.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Air Force Avionics Laboratory, in part by the Office of Naval Research under contracts N00014-84-K-0345 (N91) and N00014-84-K-0678-N123, in part by NASA under contract NCC 2-463, in part by the Army Research Institute under contract MDA903-85-C-0324, under subcontract 487650-25537 through the University of California, Irvine, and in part by small contributions from private institutions. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, ONR, NASA, ARI, or the US Government. The first and fifth authors were supported by AT&T Bell Labs Ph.D. Scholarships.

REFERENCES

1. Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers: Principles, Techniques and Tools* (Addison-Wesley, Reading, MA, 1986).
2. Anderson, J.R., Knowledge compilation: The general learning mechanism, in: *Proceedings International Machine Learning Workshop*, Montecello, IL (1983) 203–212.
3. Banerji, R.B., *Artificial Intelligence: A Theoretical Approach* (Elsevier North-Holland, New York, 1980).
4. Bennett, S.W., Approximation in mathematical domains, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 239–241.
5. Bergadano, F. and Giordana, A., A knowledge intensive approach to concept induction, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 305–317.
6. Bhatnagar, N., A correctness proof of explanation-based generalization as resolution theorem proving, in: *Proceedings AAAI Spring Symposium on Explanation-Based Learning* (1988) 220–225.
7. Braverman, M.S. and Russell, S.J., IMEX: Overcoming intractability in explanation-based learning, in: *Proceedings AAAI-88*, St. Paul, MI (1988) 575–579.

8. Brownston, L., Farrell, R., Kant, E. and Marty, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming* (Addison-Wesley, Reading, MA, 1985).
9. Carbonell, J.G., *Subjective Understanding: Computer Models of Belief Systems* (UMI Research Press, Ann Arbor, MI, 1981).
10. Carbonell, J.G., Derivational analogy and its role in problem solving, in: *Proceedings AAAI-83*, Washington, DC (1983) 64–69.
11. Carbonell, J.G., Derivational analogy: A theory of reconstructive problem solving and expertise acquisition, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986).
12. Carbonell, J.G. and Gil, Y., Learning by experimentation, in: *Proceedings Fourth International Workshop on Machine Learning*, Irvine, CA (1987) 256–266.
13. Cheng, P.W. and Carbonell, J.G., Inducing iterative rules from experience: The FERMI experiment, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 490–495.
14. Chien, S.A., Extending explanation-based learning: Failure-driven schema refinement, in: *Proceedings Third IEEE Conference on Artificial Intelligence Applications*, Orlando, FL (1987) 106–111.
15. Cohen, P.R. and Feigenbaum, E.A. (Eds.), *The Handbook of Artificial Intelligence* **3** (Kaufmann, Los Altos, CA, 1982)
16. Cohen, W.W., Generalizing number and learning from multiple examples in explanation-based learning, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 256–269.
17. Danyluk, A.P., The user of explanations for similarity-based learning, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 274–276.
18. Dean, T. and Boddy, M., An analysis of time dependent planning, in: *Proceedings AAAI-88*, St. Paul, MI (1988) 49–54.
19. DeJong, G.F., Acquiring schemata through understanding and generalizing plans, in: *Proceedings IJCAI-83*, Karlsruhe, F.R.G. (1983) 462–464.
20. DeJong, G.F. and Mooney, R., Explanation-based learning: An alternative view, *Mach. Learning* **1** (1986) 145–176.
21. DeJong, G.F., An approach to learning from observation, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 571–590.
22. DeJong, G.F., Some thoughts on the present and future of explanation-based learning, in: *Proceedings ECAI-88*, Munich, F.R.G. (1988).
23. Carbonell, J.G. and Veloso, M.M., Integrating derivational analogy into a general problem-solving architecture, Case-Based Reasoning Workshop, Clearwater Beach, FL (1988) 104–124.
24. Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979) 231–272.
25. Doyle, R.J., Constructing and refining causal explanations from an inconsistent domain theory, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 538–544.
26. Ellman, T., Generalizing logic circuit designs by analyzing proofs of correctness, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 643–646.
27. Ellman, T., Approximate theory formation: An explanation-based approach, in: *Proceedings AAAI-88*, St. Paul, MI (1988) 570–574.
28. Etzioni, O., Hypothesis filtering: A practical approach to reliable learning, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 416–429.
29. Fikes, R., Hart, P. and Nilsson, N., Learning and executing generalized robot plans, *Artificial Intelligence* **3** (1972) 251–288.
30. Flann, N.S. and Dietterich, T.G., Selecting appropriate representations for learning from examples, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 460–466.
31. Gupta, A., Explanation-based failure recovery, in *Proceedings AAAI-87*, Seattle, WA (1987) 606–610.
32. Hall, R.J., Learning by failing to explain: Using partial explanations to learn in incomplete or intractable domains, *Mach. Learning* **3** (1988) 45–77.

33. Hammond, K.J., Learning to anticipate and avoid planning problems through the explanation of failures, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 556–560.
34. Hirsh, H., Explanation-based generalization in a logic-programming environment, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 221–227.
35. Hirsh, H., Reasoning about operationality for explanation-based learning, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 214–220.
36. Kedar-Cabelli, S.T. and McCarty, L.T., Explanation-based generalization as resolution theorem proving, in: *Proceedings Fourth International Workshop on Machine Learning*, Irvine, CA (1987) 383–389.
37. Kedar-Cabelli, S.T., Formulating concepts according to purpose, in: *Proceedings AAAI-87*, Seattle, WA (1987) 477–481.
38. Keller, R.M., Learning by re-expressing concepts for efficient recognition, in: *Proceedings AAAI-83*, Washington, DC (1983) 182–186.
39. Keller, R.M., The role of explicit knowledge in learning concepts to improve performance, Ph.D. Thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ (1986).
40. Keller, R.M., Defining operationality for explanation-based learning, in: *Proceedings AAAI-87*, Seattle, WA (1987) 482–487.
41. Knoblock, C.A., Automatically generating abstractions for planning, in: *Proceedings First International Workshop in Change of Representation and Inductive Bias*, Briarcliff, NY (1988) 53–65.
42. Laird J.E., Rosenbloom, P.S. and Newell, A., Chunking in Soar: The anatomy of a general learning mechanism, *Mach. Learning* **1** (1986) 11–46.
43. Langley, P. and Carbonell, J.G., Approaches to machine learning, *J. Am. Soc. Inf. Sci.* **35** (1984) 306–316.
44. Lebowitz, M., Not the path to perdition: The utility of similarity-based learning, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 533–537.
45. Lebowitz, M., Integrated learning: Controlling explanation, *Cognitive Sci.* **10** (1986) 219–240.
46. Mahadevan, S., Verification-based learning: A generalization strategy for inferring problem-reduction methods, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 616–623.
47. Mahadevan, S., Natarajan, B. and Tadepalli, P., A framework for learning as improving problem-solving performance in: *Proceedings AAAI Spring Symposium on Explanation-Based Learning* (1988) 215–219.
48. Minton, S., Constraint-based generalization, in: *Proceedings AAAI-84*, Austin, TX (1984) 251–254.
49. Minton S., Selectively generalizing plans for problem solving, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 596–602.
50. Minton S., Carbonell, J.G., Knoblock, C.A., Kuokka, D. and Nordin, H., Improving the effectiveness of explanation-based learning, in: *Proceedings Workshop on Knowledge Compilation*, Inn at Otter Crest, OR (1986) 77–87.
51. Minton, S., Carbonell, J.G., Etzioni, O., Knoblock, C.A. and Kuokka, D.R., Acquiring effective search control rules: Explanation-based learning in the PRODIGY system, in: *Proceedings Fourth International Workshop on Machine Learning*, Irvine, CA (1987) 122–133.
52. Minton S., *Learning Search Control Knowledge: An Explanation-Based Approach* (Kluwer Academic Publishers, Boston, MA, 1988); also: Carnegie-Mellon CS Tech. Rept. CMU-CS-88-133.
53. Minton, S., EBL and weakest preconditions, in: *Proceedings AAAI Spring Symposium on Explanation-Based Learning* (1988) 210–214.
54. Mitchell, T., Utgoff, P. and Banerji, R., Learning by experimentation: Acquiring and refining problem-solving heuristics, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983) 163–190.

55. Mitchell, T.M., The need for biases in learning generalizations, Tech. Rept. CBM-TR-117, Rutgers University, New Brunswick, NJ (1980).
56. Mitchell, T., Keller, R. and Kedar-Cabelli, S., Explanation-based generalization: A unifying view, *Mach. Learning* **1** (1986) 47–80.
57. Mooney, R. and DeJong, G., Learning schemata for natural language processing, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985).
58. Mooney, R.J. and Bennett, S.W., A domain independent explanation-based generalizer, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 551–555.
59. Mooney, R.J., A general explanation-based learning mechanism and its application to narrative understanding, Ph.D. Thesis, University of Illinois at Urbana-Champaign (1987).
60. Mooney, R.J., Generalizing the order of operators in macro-operators, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 270–283.
61. Mostow, D.J., Machine transformation of advice into a heuristic search procedure, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983) 367–403.
62. Mostow, J. and Bhatnagar, N., Failsafe: A floor planner that uses EBG to learn from its failures, *Proceedings IJCAI-87*, Milan, Italy (1987) 249–255.
63. O'Rorke, P., Generalization for explanation-based schema acquisition, in: *Proceedings AAAI-84*, Austin, TX (1984) 260–263.
64. Pazzani, M., Dyer, M. and Flowers, M., The role of prior causal theories in generalization, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 545–550.
65. Pazzani, M.J., Integrated learning with incorrect and incomplete theories, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 291–297.
66. Prieditis, A.E., Discovery of algorithms from weak methods, in: *Proceedings International Meeting on Advances in Learning*, Les Arcs, Switzerland (1986) 37–52.
67. Prieditis, A.E., Environment-guided program transformation, in: *Proceedings AAAI Spring Symposium on Explanation-Based Learning* (1988) 201–209.
68. Rajamoney, S.A., and DeJong, G.F., The classification, detection and handling of imperfect theory problems, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 205–207.
69. Rosenbloom, P.S. and Laird, J.E., Mapping explanation-based generalization onto Soar, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 561–567.
70. Sacerdoti, E.D., *A Structure for Plans and Behavior* (Elsevier North-Holland, New York, 1977).
71. Schank, R.C., *Dynamic Memory* (Cambridge University Press, Cambridge, 1982).
72. Segre, A.M., Explanation-based learning of generalized robot assembly plans, Ph.D. Thesis, University of Illinois at Urbana-Champaign (1987).
73. Segre, A.M., On the operationality/generality trade-off in explanation-based learning, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 242–248.
74. Shavlik, J.W. and DeJong, G.F., An explanation-based approach to generalizing number, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 236–238.
75. Silver, B., Learning equation solving methods from worked examples, in: *Proceedings International Machine Learning Workshop*, Montecello, IL (1983).
76. Silver, B., Precondition analysis: Learning control information, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 647–670.
77. Simon, H.A., Why should machines learn? in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983) 25–37.
78. Steier, D.M., Laird, J.E., Newell, A., Rosenbloom, P.S., Flynn, R.A., Golding, A., Polk, T.A., Shivers, O.G., Unruh, A. and Yost, G.R., Varieties of learning in Soar: 1987, in: *Proceedings Fourth International Workshop on Machine Learning*, Irvine, CA (1987) 300–311.

79. Sussman, G.J., *A Computer Model of Skill Acquisition* (American Elsevier, New York, 1975).
80. Tadepalli, P., Towards learning chess combinations, Tech. Rept. ML-TR-5, Department of Computer Science, Rutgers University, New Brunswick, NJ (1986).
81. Tambe, M. and Newell, A., Some chunks are expensive, in: *Proceedings Fifth International Conference on Machine Learning*, Ann Arbor, MI (1988) 451–458.
82. Tambe, M. and Rosenbloom, P., Eliminating expensive chunks, Tech. Rept. CMU-CS-88-189, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1988).
83. Utgoff, P.E., Shift of bias for inductive concept learning, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 107–148.
84. van Harmelen, F. and Bundy, A., Explanation-based generalization = Partial evaluation, *Artificial Intelligence* **36** (1988) 401–412.
85. Vere, S.A., Splicing plans to achieve misordered goals, in: *Proceedings IJCAI-86*, Los Angeles, CA (1985) 1016–1021.
86. Waterman, D., Generalization learning techniques for automating the learning of heuristics, *Artificial Intelligence* **1** (1970) 121–170.
87. Winston, P., Learning new principles from precedents and examples, *Artificial Intelligence* **19** (1982) 321–350.