From: AAAI-86 Proceedings. Copyright ©1986, AAAI (www.aaai.org). All rights reserved.

Mapping Explanation-Based Generalization onto Soar[1]

Paul S. Rosenbloom
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
701 Welch Road (Bldg. C)
Palo Alto, CA 94304

John E. Laird
Intelligent Systems Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

ABSTRACT

Explanation-based generalization (EBG) is a powerful approach to concept formation in which a justifiable concept definition is acquired from a single training example and an underlying theory of how the example is an instance of the concept. Soar is an attempt to build a general cognitive architecture combining general learning, problem solving, and memory capabilities. It includes an independently developed learning mechanism, called chunking, that is similar to but not the same as explanation-based generalization. In this article we clarify the relationship between the explanation-based generalization framework and the Soar/chunking combination by showing how the EBG framework maps onto Soar, how several EBG concept-formation tasks are implemented in Soar, and how the Soar approach suggests answers to some of the outstanding issues in explanation-based generalization.

I INTRODUCTION

Explanation-based generalization (EBG) is an approach to concept acquisition in which a justifiable concept definition is acquired from a single training example plus an underlying theory of how the example is an instance of the concept [1, 15, 26]. Because of its power, EBG is currently one of the most actively investigated topics in machine learning [3, 5, 6, 12, 13, 14, 16, 17, 18, 23, 24, 25]. Recently, a unifying framework for explanation-based generalization has been developed under which many of the earlier formulations can be subsumed [15].

Soar is an attempt to build a general cognitive architecture combining general learning, problem solving, and memory capabilities [9]. Numerous results have been generated with Soar to date in the areas of learning [10, 11], problem solving [7, 8], and expert systems [21]. Of particular importance for this article is that Soar includes an independently developed learning mechanism, called chunking, that is similar to but not the same as explanation-based generalization.

The goal of this article is to elucidate the relationship between the general explanation-based generalization framework — as described in [15] — and the Soar approach to learning, by mapping explanation-based generalization onto Soar.[2] The resulting mapping increases our understanding of both approaches and allows results and conclusions to be transferred between them. In Sections II -IV , EBG and Soar are introduced and the initial mapping between them is specified. In Sections V and VI , the mapping is refined and detailed examples (taken from [15]) of the acquisition of a simple concept and of a search-control concept are presented. In Section VII , differences between EBG and learning in Soar are discussed. In Section VIII , proposed solutions to some of the key issues in explanation-based generalization (as set out in [15]) are presented, based on the mapping of EBG onto Soar. In Section IX , some concluding remarks are presented.

II EXPLANATION-BASED GENERALIZATION

As described in [15], explanation-based generalization is based on four types of knowledge: the goal concept, the training example, the operationality constraint, and the domain theory. The *goal concept* is a rule defining the concept to be learned. Consider the Safe-to-Stack example from [15]. The aim of the learning system is to learn the concept of when it is safe to stack one object on top of another. The goal concept is as follows.[3]

$$\neg\text{Fragile}(y) \vee \text{Lighter}(x,y) \leftrightarrow \text{Safe-to-Stack}(x,y) \quad (1)$$

The *training example* is an instance of the concept to be learned. It consists of the description of a situation in which the goal concept is known to be true. The following Safe-to-Stack training example [15] contains both relevant and irrelevant information about the situation.

On(o1,o2)
Isa(o1,box) Color(o1,Red) Volume(o1,1) Density(o1,.1) (2)
Isa(o2,endtable) Color(o2,blue)

The *operationality criterion* characterizes the generalization language; that is, the language in which the concept definition is to be expressed. Specifically, it restricts the acceptable concept descriptions to ones that are easily evaluated on new positive and negative examples of the concept. One simple operationality constraint is that the concept description must be expressed in terms of the predicates that are used to define the training example. An alternative, and the one used in [15], is to allow predicates that are used to define the training example plus other easily computable predicates, such as Less. If the goal concept meets the operationality criterion then the problem is already solved, so the cases of interest all involve a non-operational goal concept. One way to characterize EBG is as the process of operationalizing the goal concept. The goal concept is reexpressed in terms that are easily computable on the instances.

The *domain theory* consists of knowledge that can be used in proving that the training example is an instance of the goal con-

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contracts N00039-83-C-0136 and F33615-81-K-1539, and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. 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 the Defense Advanced Research Projects Agency, the US Government, the Sloan Foundation, or the National Institutes of Health.

[2]Another even more recent attempt at providing a uniform framework for explanation-based generalization can be found in [2]. It should be possible to augment the mapping described here to include this alternative view, but we do not do that here.

[3]Though this goal concept includes a pair of disjunctive clauses, only one of them actually gets used in the example.

cept. For the Safe-to-Stack example, the domain theory consists of rules and facts that allow the computation and comparison of object weights.

$$\text{Volume}(p1,v1)\wedge\text{Density}(p1,d1)\rightarrow\text{Weight}(p1,v1*d1) \quad (3)$$
$$\text{Weight}(p1,w1)\wedge\text{Weight}(p2,w2)\wedge\text{Less}(w1,w2)\rightarrow\text{Lighter}(p1,p2) \quad (4)$$
$$\text{Isa}(p1,\text{endtable})\rightarrow\text{Weight}(p1,5) \quad (5)$$
$$\text{Less}(.1,5) \quad (6)$$
...

Given the four types of knowledge just outlined, the EBG algorithm consists of three steps: (1) use the domain theory to prove that the training example is an instance of the goal concept; (2) create an explanation structure from the proof — the tree structure of rules that were used in the proof — filtering out rules and facts that turned out to be irrelevant; and (3) regress the goal concept through the explanation structure — stopping when operational predicates are reached — to yield the general conditions under which the explanation structure is valid. The desired concept definition consists of the conditions generated by the regression process.

$$\text{Volume}(x,v)\wedge\text{Density}(x,d)\wedge\text{Isa}(y,\text{endtable})\wedge\text{Less}(v*d,5)\rightarrow\text{Safe-to-Stack}(x,y) \quad (7)$$

III SOAR

The most complete description of Soar can be found in [9], and summaries can be found in most of the other Soar articles. Without going into great detail here, the mapping of explanation-based generalization onto Soar depends upon five aspects of the Soar architecture.

Problem spaces. Problem spaces are used for all goal-based behavior. This defines the deliberate acts of the architecture: selection of problem spaces, states, and operators.

Subgoals. Subgoals are generated automatically whenever an impasse is reached in problem solving. These impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of operator instantiation and application. When subgoals occur within subgoals, a goal hierarchy results. An object created in a subgoal is a result of the subgoal if it is accessible from any of the supergoals, where an object is accessible from a supergoal if there is a link from some other object in the supergoal to it (this is all finally rooted in the goals).

Production systems. A production system is used as the representation for all long-term knowledge, including factual, procedural, and control information. The condition language is limited to the use of constants and variables, the testing of equality and inequality of structured patterns, and the conjunction of these tests. Disjunction is accomplished via multiple productions. The action language is limited to the creation of new elements in the working memory — functions are not allowed. The working memory is the locus of the goal hierarchy and of temporary declarative information that can be created and examined by productions.

Decision Cycle. Each deliberate act of the architecture is accomplished by a decision cycle consisting of a monotonic elaboration phase, in which the long-term production memory is accessed in parallel until quiescence, followed by a decision procedure which makes a change in the problem-solving situation based on the information provided by the elaboration phase. All of the control in Soar occurs at this problem-solving level, not at the level of (production) memory access.

Chunking. Productions are automatically acquired that summarize the processing in a subgoal. The actions of the new productions are based on the results of the subgoal. The conditions are based on those aspects of the initial situation that were relevant to the determination of those results.

IV THE INITIAL MAPPING

Given the descriptions of explanation-based generalization and Soar, it is not difficult to specify an initial mapping of EBG onto Soar (Figure 1). The goal concept is simply a goal to be achieved. The training example is the situation that exists when a goal is generated. The operationality criterion is that the concept must be a production condition pattern expressed in terms of the predicates existing prior to the creation of the goal (a disjunctive concept can be expressed by a set of productions). The domain theory corresponds to a problem space in which the goal can be attempted, with the predicates defined by the theory corresponding to operators in the problem space.

EBG		Soar
goal concept	⇔	goal
training example	⇔	pre-goal situation
operationality constraint	⇔	pre-goal predicates
domain theory	⇔	problem space

Figure 1: The mapping of the EBG knowledge onto Soar.

The Safe-to-Stack problem can be implemented in Soar by defining an operator, let's call it Safety?(x, y), which examines a state containing two objects, and augments it with information about whether it is safe to stack the first object on the second object. The concept will be operational when a set of productions exist that directly implement the Safety? operator. When the operator is evoked and no such productions exist — that is, when the operational definition of the concept has not yet been learned — an operator-implementation subgoal is generated because Soar is unable to apply the Safety? operator. In this subgoal the domain-theory problem space can be used to determine whether it is safe to stack the objects. On the conclusion of this problem solving, chunks are learned that operationalize the concept for some class of examples that includes the training example. Future applications of the Safety? operator to similar examples can be processed directly by the newly acquired productions without resorting to the domain theory.

V A DETAILED EXAMPLE

In this section we take a detailed look at the implementation of the Safe-to-Stack problem in Soar, beginning with the training example, the domain theory, and the goal concept; followed by a description of the concept acquisition process for this task.

The standard way to represent objects in Soar is to create a temporary symbol, called an *identifier*, for each object. All of the information about the object is associated with its identifier, including its name. This representational scheme allows object identity to be tested — by comparing identifiers — without testing object names, a capability important to chunking. In this representation, the Safe-to-Stack training example involves a slightly more elaborated set of predicates than is used in (2). The identifiers are shown as greek letters. In (2), the symbols o1 and o2 acted like identifiers. In the example below they are replaced by α and β respectively.

$$\text{On}(\alpha,\beta)$$

$$\text{Name}(\alpha,\text{box})\ \ \text{Color}(\alpha,\gamma)\ \ \text{Volume}(\alpha,\delta)\ \ \text{Density}(\alpha,\varepsilon)\ \ \text{Name}(\gamma,\text{red})\ \ \text{Name}(\delta,1)\ \ \text{Name}(\varepsilon,.1) \quad (8)$$

$$\text{Name}(\beta,\text{endtable})\ \ \text{Color}(\beta,\zeta)\ \ \text{Name}(\zeta,\text{blue})$$

The domain theory is implemented in Soar by a problem space, called *Safe*, containing the following four operators. Each of these operators is implemented by one or more productions. In production number 9, the use of a variable (w) in the action of the

production, without it appearing in a condition, denotes that a new identifier is to be created and bound to that variable.

Weight?(*p*)
Name(p,endtable) → Weight(p,w)∧Name(w,5) (9)
Volume(p,v)∧Density(p,d)∧Product(v,d,w) → Weight(p,w) (10)
Lighter?(*p1*,*p2*)
Weight($p1,w1$)∧Weight($p2,w2$)∧Less($w1,w2$) → Lighter($p1,p2$) (11)
Less?(*n1*,*n2*)
Name($n1$,.1)∧Name($n2$,5) → Less($n1,n2$) (12)
...
Product?(*n1*,*n2*)
Name($n1$,1) → Product($n1,n2,n2$) (13)
...

There are two notable differences between this implementation and the domain theory specified in EBG. The first difference is that the Less predicate is defined by an operator rather than simply as a set of facts. Because all long-term knowledge in Soar is encoded in productions, this is the appropriate way of storing this knowledge. The implementation of the operator consists of one production for each fact stored (it could also have been implemented as a general algorithm in an operator-implementation subgoal). The second difference is that the multiplication of the volume by the density is done by an operator rather than as an action in the right-hand side of the production. Because Soar productions do not have the capability to execute a multiplication operation as a primitive action, Product needs be handled in a way analogous to Less. For this case we have provided one general production which knows how to multiply by one. To implement the entire Product predicate would require either additional implementation productions or a general multiplication problem space in which a multiplication algorithm could be performed.

The Safe problem space also contains one more operator that defines the goal concept. For the purposes of problem solving with this information, the rule defining the goal concept need not be treated differently from the rules in the domain theory. It is merely the last operator to be executed in the derivation.

Safe-to-Stack?(*p1*,*p2*)
Fragile($p2$) → Safe-to-Stack($p1,p2$) (14)
Lighter($p1,p2$) → Safe-to-Stack($p1,p2$) (15)

In addition to the operators, there are other productions in the Safe problem space that create the initial state and propose and reject operators.[4] In what follows we will generally ignore these productions because they add very little to the resulting concept definition — either by not entering into the explanation structure (the reject productions), having conditions that duplicate conditions in the operator productions (the propose productions), or adding general context-setting conditions (the initial-state production). In other tasks, these productions can have a larger effect, but they did not here.

Figure 2 shows the mapping of the EBG process onto Soar. The EBG proof process corresponds in Soar to the process of problem solving in the domain-theory problem space, starting from an initial state which contains the training example, and terminating when a state matching the goal concept is achieved.

We can pick up the problem solving at the point where there is a state selected that contains the training example and a Safety? operator selected for that state. Because the operator cannot be directly applied to the state, an operator-implementation subgoal is immediately generated, and the Safe problem space is selected for this new goal. If there is enough search-control knowledge available to uniquely determine the sequence of operators to be selected and applied (or outside guidance is provided), so that operator-selection subgoals are not needed, then the following sequence of operator instances, or one functionally equivalent to it, will be selected and applied.

EBG		Soar
proof	⇔	problem solving
explanation structure	⇔	backtraced production traces
goal regression	⇔	backtracing and variablization

Figure 2: The mapping of the EBG process onto Soar.

Weight?(β)
Name(β,endtable) → Weight(β,η)∧Name(η,5) (16)
Product?(δ,ε)
Name(δ,1) → Product($\delta,\varepsilon,\varepsilon$) (17)
Weight?(α)
Volume(α,δ)∧Density(α,ε)∧Product($\delta,\varepsilon,\varepsilon$) → Weight($\alpha,\varepsilon$) (18)
Less?(ε,η)
Name(ε,.1)∧Name(η,5) → Less(ε,η) (19)
Lighter?(α,β)
Weight(α,ε)∧Weight(β,η)∧Less(ε,η) → Lighter(α,β) (20)
Safe-to-Stack?(α,β)
Lighter(α,β) → Safe-to-Stack(α,β) (21)

As they apply, each operator adds information to the state. The final operator adds the information that the Safety? operator in the problem space above was trying to generate — Safe-to-Stack(α, β). A test production detects this and causes the Safety? operator to be terminated and the subgoal to be flushed.

After the subgoal is terminated, the process of chunk acquisition proceeds with the creation of the explanation structure. In Soar, this is accomplished by the architecture performing a backtrace over the production traces generated during the subgoal. Each production trace consists of the working-memory elements matched by the conditions of one of the productions that fired plus the working-memory elements generated by the production's actions. The backtracing process begins with the results of the subgoal — Safe-to-Stack(α, β) — and traces backwards through the production traces, yielding the set of production firings that were responsible for the generation of the results. The explanation structure consists of the set of production traces isolated by this backtracing process.[5] In the Safe-to-Stack example, there is only one result, and its explanation structure consists of the production traces listed above (productions 16-21). Other productions have fired — to propose and reject operators, generate the initial state in the subgoal, and so on — but those that did enter the explanation structure only added context-setting conditions, linking the relevant information to the goal hierarchy and the current state. They did not add any conditions that test aspects of the training example.

The backtracing process goes a step beyond determining the explanation structure. It also determines which working-memory elements should form the basis of the conditions of the chunk by

[4] There are no search-control productions in this implementation of the Safe-to-Stack problem (except for ones that reject operators that are already accomplished). Instead, for convenience we guided Soar through the task by hand. The use of search-control productions would not change the concept learned because Soar does not include them in the explanation structure [9].

[5] It is worth noting that earlier versions of Soar computed the conditions of chunks by determining which elements in working memory were examined by any of the productions that executed in the subgoal [11]. When the problem solving is constrained to look only at relevant information, as it was in the early work on human practice [20], this worked fine. However, in a system that is searching, often down what turn out to be dead ends, this assumption can be violated, leading to chunks that are overspecific. Backtracing was added to Soar to avoid these problems with dead ends [9, 11]. This modification was based on our understanding of the EBG approach, but it was not done directly to model EBG.

isolating those that are (1) part of the condition side of one of the production traces in the explanation structure and (2) existed prior to the generation of the subgoal. The actions of the chunk are based directly on the goal's results. The following instantiated production is generated by this process.

Safety?(α,β)

$$\text{Volume}(\alpha,\delta)\wedge\text{Name}(\delta,1)\wedge\text{Density}(\alpha,\varepsilon)\wedge\text{Name}(\varepsilon,.1)\wedge\text{Name}(\beta,\text{endtable})\rightarrow\text{Safe-to-Stack}(\alpha,\beta) \quad (22)$$

Soar's condition-finding algorithm is equivalent to regressing the instantiated goal — actually, the results of the goal — through the (instantiated) production traces.[6] It differs from the EBG goal regression process in [15] in making use of the instantiated goal and rules rather than the more general parameterized versions. The Soar approach to goal regression is simpler, and focuses on the information in working memory rather than the possibly complex patterns specified by the rules, but it does not explain how variables appear in the chunk. Variables are added during a later step by replacing all of the object identifiers with variables. Identifiers that are the same are replaced by the same variable and identifiers that are different are replaced by different variables that are forced to match distinct objects. The following variablized rule was generated by Soar for this task.

Safety?(x,y)

$$\text{Volume}(x,v)\wedge\text{Name}(v,1)\wedge\text{Density}(x,d)\wedge\text{Name}(d,.1)\wedge\text{Name}(y,\text{endtable})\rightarrow\text{Safe-to-Stack}(x,y) \quad (23)$$

This production is not as general as the rule learned by EBG (rule 7 in Section II). Instead of testing for the specific volume and density of the first object, the EBG rule tests that their product is less than 5. This happens because the EBG implementation assumed that Less and Product were operational; that is, that they were predicates at which regression should stop. In the Soar example they were not operational. Both of these predicates were implemented by operators, so the regression process continued back through them to determine what lead them to have their particular values.

In EBG, operationalized predicates showed up in one of two ways: either the set of true instances of the predicate was included along with the domain theory and the training example (Less), or the predicate was a primitive operation in the rule language (Product). The key point about both of these approaches is that the computation of the value of the predicate will be cheap, not requiring the use of inference based on rules in the domain theory. In Soar, any preexisting predicate is cheap, but functions are not allowed in the rule language. Therefore, the way for Soar to generate the more general rule is to make sure that all of the operational predicates preexist in working memory. Specifically, objects representing the numbers and the predicates Less and Product need to be available in working memory before the subgoal is generated, and the endtable-weight rule must be changed so that it makes use of a preexisting object representing the number 5 rather than generating a new one. Under these circumstances, backtracing stops at these items, and the following production is generated by Soar.

Safety?(x,y)

$$\text{Volume}(x,v)\wedge\text{Density}(x,d)\wedge\text{Name}(y,\text{endtable})\wedge\text{Product}(v,d,d)\wedge\text{Less}(d,w)\wedge\text{Name}(w,5)\rightarrow\text{Safe-to-Stack}(x,y) \quad (24)$$

This production does overcome the problem, but it is still more specific than rule 7 because the density and the weight of the box were the same in this example (they were represented by the same identifier) — the variablization strategy avoids creating overly general rules but can err in creating rules that are too specific. Thus the chunk only applies in future situations in which this is true. If an example were run in which the density and the weight were different, then a rule would be learned to deal with future situations in which they were different.

[6] See [19] for a good, brief description of goal regression.

VI SEARCH-CONTROL CONCEPTS

One of the key aspects of Soar is that different types of subgoals occur in different situations. The implication of this for EBG is that the type of subgoal determines the type of concept to be acquired. In the previous section we have described concept formation based on one type of subgoal: operator implementation. In this section we look at one other type of subgoal, operator selection, and show how it can lead to the acquisition of search-control concepts — descriptions of the situations in which particular operators have particular levels of utility. The process of extending the mapping between EBG and Soar to this case reveals the underlying relationships among the various types of knowledge and processes used in the acquisition of search-control concepts.

As described in [15], in addition to the four types of knowledge normally required for EBG, its use for the acquisition of search-control concepts requires two additional forms of knowledge: (1) the *solution property*, which is a task-level goal; and (2) the *task operators*, which are the operators to be used in achieving the solution property. For example, in the symbolic integration problem posed in [15], the solution property is to have an equation without an integral in it, and the task operators specify transformation rules for mathematical equations.

The domain theory includes task-dependent rules that determine when a state is *solved* (the solution property is true) and task-independent rules that determine whether a state is *solvable* (there is a sequence of operators that will lead from the state to a solved state) and specify how to regress the solution property back through the task operators. The goal concept is to determine when a particular task operator — Op3, which moves a numeric coefficient outside of the integral in forms like $\int 7x^2dx$ — is useful in achieving the solution property. That is, we are looking for a description of the set of unsolved states for which the application of the operator leads to a solvable state.

The EBG approach to solving this generalization problem involves two phases, both of which are controlled by the domain theory. In the first phase, a search is performed with the task operators to determine whether the state resulting from the application of Op3 is solvable. In the second phase, the solution property is regressed through the task operators in the solution path — a deliberate regression controlled by the domain theory, not the regression that automatically happens with EBG. These steps form the basis for the explanation structure. Using a training example of $\int 7x^2dx$, the following concept description was learned [15].

$$\text{Matches}(y,f(\cdot)\int r\cdot x^s dx)\wedge\text{Isa}(r,\text{real})\wedge\text{Isa}(s,\text{real})\wedge\text{Isa}(f,\text{function})\wedge\text{Not-Equal}(s,-1)\rightarrow\text{Useful-Op3}(y) \quad (25)$$

In the Soar implementation of this task, the solution property and task operators correspond to a goal and problem space respectively. The task-level search to establish state solvability corresponds to a search in this problem space. The regression of the solution property through the task operators simply corresponds to chunking. In fact, all of the knowledge and processing required for this task map cleanly into a hierarchy of subgoals in Soar, as shown in Figure 3. The details of this mapping should become clear as we go along.

The problem solving that generates these goals is shown in Figure 4. At the very top of the figure is the task goal of having an integral-free version of the formula. This first goal corresponds to the solution property in the EBG formalism (Figure 3). A problem space containing the task operators is used for this goal and the initial state represents the formula to be in-

EBG		Soar
solution property	⇔	Goal 1: problem-solving task
goal concept	⇔	Goal 2: operator selection
domain theory	⇔	Goal 3: operator evaluation
solved	⇔	goal test
solvable	⇔	problem solving
regression	⇔	chunking
task operators	⇔	Problem space for goals 1 & 3

Figure 3: Extending the mapping for search-control concepts.

tegrated. Because both operator Op3 and another operator are acceptable for this state, and the knowledge required to choose between them is not directly available in productions, an operator-selection subgoal is created. This second subgoal corresponds to the EBG goal concept (Figure 3) — the desire to determine the knowledge necessary to allow the selection of the most appropriate operator. In this subgoal, search-control knowledge about the utility of the competing operators (for the selected state) is generated until an operator can be selected. For such goals, Soar normally employs the *Selection* problem space. The Selection problem space contains an Evaluate operator, which can be applied to the competing task operators to determine their utility.

If, as is often the case, the information about how to evaluate an operator is not directly available, an evaluation subgoal (to implement the Evaluate operator) is created. The task in this third-level subgoal is to determine the utility of the operator. To do this, Soar selects the original task problem space and state, plus the operator to be evaluated. It then applies the operator to the state, yielding a new state. If the new state can be evaluated, then the subgoal terminates, otherwise the process continues,

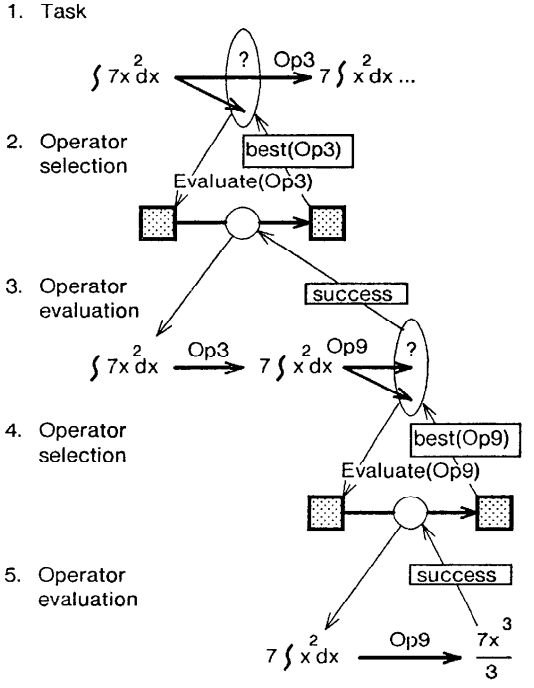

Figure 4: Problem solving in the symbolic integration task.

generating further levels of selection and evaluation goals, until a task desired state — that is, a state matching the solution property — is reached, or the search fails. For this problem, the search continues until Op9 — which integrates equations of the form $\int x^s dx$ — is applied. At this point Op3 is given an evaluation of *success*, resulting in search-control knowledge being generated that says that no other operator will be better than it (similar processing also occurs for Op9). Because this new knowledge is sufficient to allow the selection of Op3 in the top goal, it gets selected and applied immediately, terminating the lower goals.

The EBG domain theory maps onto several aspects of the processing of the third-level evaluation subgoal (Figure 3). The EBG rules that determine when a state is solved correspond to a goal test rule. The EBG rules that determine state solvability correspond to the problem-solving strategy in the evaluation subgoal. The EBG rules that deliberately regress the solution property through the task operators correspond to the chunking process on the evaluation subgoal — goal regression in Soar is thus always done by chunking, but possibly over different subgoals. These chunks are included as part of the explanation structure for the parent operator-selection goal because the processing in the bottom subgoal was part of what lead to the parent goal's result.

Chunks are learned for each of the levels of goals, but the ones of interest here are for the operator-selection subgoals. These chunks provide search-control knowledge for the task problem space — the focus of this section. Soar acquired the following production for the top operator-selection goal. This production is presented in the same abstract form that was used for the corresponding EBG rule (rule 25).

$$\text{Proposed}(a) \wedge \text{Name}(a,\text{Op3}) \wedge \text{Integral}(a,b) \wedge \text{Matches}(b, \int r \cdot x^s dx) \wedge \text{Isa}(r,\text{real}) \wedge \text{Isa}(s,\text{real}) \wedge \text{Not-Equal}(s,-1) \rightarrow \text{Best}(a) \quad (26)$$

This production specifies a class of situations in which operator Op3 is best.[7] Operator Op3 takes three parameters in the Soar implementation: (1) the integral ($\int 7x^2 dx$); (2) the coefficient (7); and (3) the term (x^2), which is the other multiplicand within the integral. The predicates in this production examine aspects of these arguments and their substructures. Though the details tend to obscure it, the content of this production is essentially the same as the EBG rule.

Two additional points raised by this example are worth mentioning. The first point is that the EBG rule (rule 25) checks whether there is an arbitrary function before the integral, whereas this rule does not. The additional test is absent in the Soar rule because the representation — which allowed both tree structured and flat access to the terms of the formula — allowed any term to be examined and changed independent of the rest of the formula. Functions outside of the integral are thus simply ignored as irrelevant. The second point is that the learning of this rule requires the climbing of a type hierarchy. The training example mentions the number 7, but not that it is a real number. In Soar, the type hierarchy is defined by adding a set of rules which successively augment objects with higher-level descriptors. All of these productions execute during the first elaboration phase after the training example is defined, so the higher-level descriptors are already available — that is, operational — before the subgoal is generated. This extra knowledge about the semantics of the concept description language is thus encoded uniformly in the problem solver's memory along with the rest of the system's knowledge.

[7] To Soar, stating that an object is best means that the object is at least as good as any other possibility. Because operators are only being rated here on whether their use leads to a goal state, best here is equivalent to useful in rule 25.

VII DIFFERENCES IN THE MAPPING

The previous sections demonstrate that the EBG framework maps smoothly onto Soar, but three noteworthy differences did show up. The first difference is that EBG regresses a variablized goal concept back through variablized rules, whereas chunking regresses instantiated goal results through instantiated rules (and then adds variables). In other words, both schemes use the explanation structure to decide which predicates from the training example get included in the concept definition — thus placing the same burden on the representation in determining the generality of the predicates included[8] — but they differ in how the definition is variablized. In EBG, this process is driven by unification of the goal concept with the relevant rules of the domain theory, whereas in chunking it is driven by the representation of the training example (that is, which identifiers appear where). Putting more of the burden on the representation allows the chunking approach to be more efficient, but it can also lead to the acquisition of overly-specific concept definitions.

The second difference is that predicates can be operationalized in EBG either by including them as facts in the domain theory or by making them into built-in functions in the rule language. In Soar, only the predicates existing in working memory prior to the generation of a subgoal are operational for that subgoal. This is not a severe limitation because any predicate that can be implemented in the rule language can also be implemented by an operator in Soar, but it could lead to efficiency differences. One direction that we are actively pursuing is the dynamic augmentation of the set of operational predicates for a goal concept during the process of finding a path from the training example to the goal concept. If intermediate predicates get operationalized — that is chunks are learned for them — then the overall goal concept can be expressed in terms of them rather than just the preexisting elements.[9]

The third difference is that the EBG implementation of search-control acquisition requires the addition of general interpretive rules to enable search with the task operators and the regression of the solution property through them[10], while Soar makes use of the same goal/problem-space/chunking approach as is used for the rest of the processing. In the Soar approach, the representation is uniform, and the different components integrate together cleanly.

VIII EBG ISSUES

In [15], four general issues are raised about EBG:

1. The use of imperfect domain theories.
2. The combination of explanation-based and similarity-based methods.
3. The formulation of generalization tasks.
4. The use of contextual knowledge.

The purpose of this section is to suggest solutions to three of these issues — the second issue has not yet been seriously investigated in Soar, so rather than speculate on how it might be done, we will leave that topic to a later date. Other solutions have been suggested for these issues (see [15] for a review of many of these), but the answers presented here are a mutually compatible set derived from the mapping between EBG and Soar.

The first issue — the use of imperfect domain theories — arises because, as specified in [15], in order to use EBG it is necessary to have a domain theory that is (1) complete, (2) consistent, and (3) tractable. Should any of these conditions not hold, it will be impossible to prove that the training example is an instance of the concept. Not mentioned in [15], but also important, is that the correctness of the resulting generalization is influenced by two additional conditions on the domain theory: that it be (4) free of errors and (5) not defeasible (a defeasible domain theory is one in which the addition of new knowledge can change the outcome).

If the process of generating an explanation is viewed not as one of generating a correct proof, but of solving a problem in a problem space, then the first four conditions reduce to the same ones that apply to any problem solver. Moreover they are all properties of applying the problem space to individual problems, and not of the problem space (or domain theory) as a whole. A space that is perfectly adequate for a number of problems may fail for others. As such, violations of these conditions can be dealt with as they arise on individual problems. In Soar, varying amounts of effort have gone into investigating how to deal with violations of these conditions. A variety of techniques have been used to make problem solving more tractable, including chunking, evaluation functions, search-control heuristics, subgoals, and abstraction planning [9, 10, 11, 21, 22]. However, the other conditions have been studied to a much lesser extent.

The one condition that does not involve a pure problem solving issue is defeasibility. The explanation process may progress smoothly and without error with a defeasible theory, but it can lead to overgeneralization in both EBG and Soar. In the EBG version of the Safe-to-Stack problem, the theory is defeasible because, as specified in [15], the rule which computes the weight of the endtable (rule 5) is actually a default rule which can be overridden by a known value. The acquired concept definition (rule 7) is thus overgeneral. It will incorrectly apply in situations where there is a small non-default weight for the endtable. In Soar, domain theories can be defeasible for a number of reasons, including the use of default processing for the resolution of impasses and the use of negated conditions in productions.[11] Sometimes the domain theory can be reformulated so that it is not defeasible, and at other times it is possible to reflect the defeasibility of the domain theory in the concept definition — for example, by including negated conditions in the concept definition — but when defeasibility does exist and yields overgeneralization, the problem of recovering from overgeneralization becomes key. Though we do not have this problem completely solved, Soar can recover when an overgeneral chunk fails to satisfy some higher goal in the hierarchy.

As mentioned in [15], the third issue — the formulation of generalization problems — is resolved by Soar. Whenever a subgoal is generated, a generalization problem is implicitly defined. The subgoal is a problem-solving goal — to derive the knowledge that will allow problem solving to continue — rather than a learning goal. However, one of the side effects of subgoal processing is the creation of new chunk productions which encode the generalized relationship between the initial situation and the results of the subgoal.

The fourth issue — the use of contextual knowledge — is straightforward in Soar. At each decision, all of the knowledge available in the problem space that is relevant to the current situation is accessed during the elaboration phase. This can include general background and contextual knowledge as well

[8] See [11] for a discussion of the interaction between representation and generality in Soar.

[9] The approach is not unlike the scheme independently developed by DeJong and Mooney [2].

[10] However, a more uniform approach to the acquisition of search-control rules by EBG can be developed [4].

[11] Negated conditions test for the absence of an element of a certain type, which is not the same as testing whether the negation of an element is known to be true (as is done by the Not-Equal predicate in production 26).

as more local knowledge about the task itself.

IX CONCLUSION

Explanation-based generalization and Soar/chunking have been described and related, and examples have been provided of Soar's performance on two of the problems used to exemplify EBG in [15]. The mapping of EBG onto Soar is close enough that it is safe to say that chunking is an explanation-based generalization method. However, there are differences in (1) the way goal regression is performed, (2) the locus of the operational predicates, and (3) the way search-control concepts are learned.

Mapping EBG onto Soar suggests solutions to a number of the key issues in explanation-based generalization, lending credence to the particular way that learning and problem solving are integrated together in Soar. Also, based on the previous experience with Soar in a variety of tasks [9] — including expert-system tasks [21] — this provides evidence that some form of EBG is widely applicable and can scale up to large tasks.

ACKNOWLEDGMENTS

We would like to thank Tom Dietterich, Gerald DeJong, Jack Mostow, Allen Newell, and David Steier, for their helpful comments on drafts of this article. We would also like to thank the members of the Soar and Grail groups at Stanford for their feedback on this material.

REFERENCES

1. DeJong, G. Generalizations based on explanations. Proceedings of IJCAI-81, 1981.

2. DeJong, G., & Mooney, R. "Explanation-based learning: An alternative view." *Machine Learning 1* (1986). In press

3. Ellman, T. Explanation-based learning in logic circuit design. Proceedings of the Third International Machine Learning Workshop, Skytop, PA, 1985, pp. 35-37.

4. Hirsh, H. Personal communication. 1986

5. Kedar-Cabelli, S. T. Purpose-directed analogy. Proceedings of the Cognitive Science Conference, Irvine, CA, 1985.

6. Keller, R. M. Learning by re-expressing concepts for efficient recognition. Procedings of AAAI-83, Washington, D.C., 1983, pp. 182-186.

7. Laird, J. E. *Universal Subgoaling*. Ph.D. Th., Carnegie-Mellon University, 1983.

8. Laird, J. E., and Newell, A. A universal weak method: Summary of results. Proceedings of the Eighth IJCAI, 1983.

9. Laird, J. E., Newell, A., & Rosenbloom, P. S. Soar: An architecture for general intelligence. In preparation

10. Laird, J. E., Rosenbloom, P. S., & Newell, A. Towards chunking as a general learning mechanism. Proceedings of AAAI-84, Austin, 1984.

11. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The anatomy of a general learning mechanism." *Machine Learning 1* (1986).

12. Lebowitz, M. Concept learning in a rich input domain: Generalization-based memory. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, Eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

13. Mahadevan, S. Verification-based learning: A generalization strategy for inferring problem-decomposition methods. Proceedings of IJCAI-85, Los Angeles, CA, 1985.

14. Minton, S. Constraint-based generalization: Learning game-playing plans from single examples. Proceedings of AAAI-84, Austin, 1984, pp. 251-254.

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

16. Mitchell, T. M., Mahadevan, S., & Steinberg, L. LEAP: A learning apprentice for VLSI design. Proceedings of IJCAI-85, Los Angeles, CA, 1985.

17. Mitchell, T. M., Utgoff, P. E., & Banerji, R. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983, pp. 163-190.

18. Mooney, R. Generalizing explanations of narratives into schemata. Proceedings of the Third International Machine Learning Workshop, Skytop, PA, 1985, pp. 126-128.

19. Nilsson, N.. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

20. Rosenbloom, P. S., & Newell, A. The chunking of goal hierarchies: A generalized model of practice. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, Eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

21. Rosenbloom, P. S., Laird, J. E., McDermott, J., & Orciuch, E. "R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture." *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 561-569.

22. Rosenbloom, P. S., Laird, J. E., Newell, A., Golding, A., & Unruh, A. Current research on learning in Soar. Proceedings of the Third International Machine Learning Workshop, Skytop, PA, 1985, pp. 163-172.

23. Segre, A. M. Explanation-based manipulator learning. Proceedings of the Third International Machine Learning Workshop, Skytop, PA, 1985, pp. 183-185.

24. Tadepalli, P. Learning in intractable domains. Proceedings of the Third International Machine Learning Workshop, Skytop, PA, 1985, pp. 202-205.

25. Utgoff, P. E. Adjusting bias in concept learning. Proceedings of IJCAI-83, Karlsruhe, West Germany, 1983, pp. 447-449.

26. Winston, P. H., Binford, T. O., Katz, B., & Lowry, M. Learning physical descriptions from functional definitions, examples, and precedents. Proceedings of AAAI-83, Washington, 1983, pp. 433-439.