Smith, R.G., Mitchell, T.M., Chestek, R.A., & Buchanan, B.G. (1977). A model for learning systems. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.* (pp. 338 – 343). Cambridge, Mass.: Morgan Kaufmann.

Sussman, G.J. (1977). *A computer model of skill acquisition.* New York: Elsevier.

Utgoff, P.E. (1984). *Shift of bias for inductive concept learning.* Doctoral dissertation, Rutgers University, New Brunswick, NJ.

van de Brug, A., Rosenbloom, P.S., & Newell, A. (1985). *Some experiments with R1-Soar* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA. In preparation.

Waterman, D.A. (1975). Adaptive production systems. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 296 – 303). Tbilisi, USSR: Morgan Kaufmann.

CHAPTER 11

Overgeneralization During Knowledge Compilation in Soar[1]

J. E. Laird, P. S. Rosenbloom[2], and A. Newell[3]

Soar is an attempt to realize in a single system a set of hypotheses on the nature of general intelligence [5]. One central hypothesis is that chunking [20], a form of knowledge compilation, is sufficient for a wide class of learning tasks. We have previously demonstrated its ability to learn a variety of types of knowledge, work in a variety of domains and transfer what it learns to new tasks [6,7]. In this article we consider the issue of overgeneralization; that is, the occurrence of performance errors as the result of the inappropriate application of compiled knowledge to situations other than the one for which it was learned. We examine this problem for knowledge-compilation systems in general, and Soar in particular. In the following sections we give an overview of the Soar architecture, describe the learning in Soar in terms of knowledge compilation, examine the ways overgeneralization arises, and discuss how it is possible to avoid or recover from overgeneralization.

Soar

The most complete descriptions of Soar can be found in [4] and [5]. In this section we provide a summary of those aspects of Soar relevant to knowledge compilation.

Problem spaces. Problem spaces are used for all goal-oriented behavior. The problem space determines the set of legal states and operators that can be used during the processing to attain the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The problem solving process consists of selecting problem spaces, states and operators, applying operators to states and detecting that a newly generated state achieves the goal.

Production Systems. A production system is used as the representation for all long-term knowledge, including factual, procedural (operator application), and control information. The production system's working memory contains temporary processing information about the system's goals, problem spaces,

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contracts N00039-83-C-0136 and F33615-81-K-1539. 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 or the US Government.

[2]Departments of Computer Science and Psychology, Stanford University.

[3]Department of Computer Science, Carnegie-Mellon University.

states and operators. The conditions of the productions test for the presence or absence of elements in working memory, while the actions of productions add elements to working memory. Removal is handled automatically by a working-memory manager (see below).

Decision Cycle. Each deliberate act of the Soar architecture (a selection of a problem space, state or operator) is accomplished by a decision cycle consisting of a monotonic elaboration phase, in which the long-term memory is accessed (i.e., productions are fired) in parallel until quiescence (i.e., until no more productions can fire), followed by a decision that causes a change to be made in the problem-solving context. The decision is based on *preferences* generated during the elaboration phase (and placed in working memory). There is a fixed language of preferences which is used to describe the acceptability and desirability of the alternatives being considered for selection. All of the control in Soar occurs at this problem-solving level, not at the level of (production) memory access (there is no conflict resolution process in Soar's production system).

Subgoals. While working on a goal, if the preferences for a decision are either incomplete or inconsistent, an impasse occurs in problem solving (i.e., the system does not know how to make progress on that goal). When an impasse occurs, a subgoal is automatically generated for the task of resolving the impasse. The impasses and thus their subgoals, vary from problems of selection (of problem spaces, states and operators) to problems of generation (e.g., operator application). A subgoal terminates when its impasse is resolved. On termination, all subgoals of the terminated subgoal are also terminated. All of the working-memory elements of a terminated subgoal are deleted by the *working-memory manager* except for those that are results of the subgoal.

Chunking. Productions (referred to as *chunks)* 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 of a chunk are based on the aspects of the pre-impasse working memory that were tested by productions that contributed, directly or indirectly, to the creation of the results that became the actions of the chunk.

Knowledge Compilation

For present purposes, we define knowledge compilation to be a conversion of knowledge from one representation to another, more efficient, form. Examples of this type of knowledge compilation include memo functions [10,12], production composition [9,16], explanation-based learning [14] and chunking in Soar. The target language of the compilation may be the same as or different from the source language - the language of the original representation of the knowledge. In this article will we concentrate on the compilation of knowledge that is originally encoded as a process, such as a procedure, a method, or the problem solving to achieve a goal.

We propose the following definition for this form of knowledge compilation.

> Let p_1 be a process that converts one situation into another, $p_1(s)=>s'$. The domain of the process is the set of all situations to which it can be applied and the range is the set of all situations that it can produce, p_1: S -> S'. Knowledge compilation (KC) takes a specification of a process and its domain and produces a new, compiled process, $KC(p_1, S) => p_2$. Process p_2 is then used in place of p_1 for some subset of the situations of S, called S_{p2} and for all s_{p2} that are elements of S_{p2}, $p_2(s_{p2}) \sim p_1(s_{p2})$, where ~ means acceptably the same.

For a process that is embedded in a larger system, the situation it processes is the global state of the overall system before the process is run. The situation that results from the application of p_1 to s, s', is the global state of the system after the process has executed. In a Lisp system, where a process would be a function, a situation would include the complete state of the Lisp, which includes but is not limited to the arguments to the function, all global variables, and the control stack.

One property of many processes is that they ignore many aspects of a situation and only make small changes to create a new situation. For a given application of process, $p_1(s) => s'$, we define that part of s' that differs from s to be the output of p_1, and we define that part of s that was necessary to produce the output to be the input of p_1. In a Lisp system, the input to a function would be the function's parameters, those global variables tested in the function (and any subfunctions), and possibly the control stack. The output would be the result of the function and any changes to global variables, the control stack or the external environment.

As Mostow and Cohen observed in their work on memoizing Lisp [15], knowledge compilation can adversely affect the behavior of the total system unless the results of the compiled process for a given input are *acceptably the same* as the results of the original process. Acceptability is specific to the implementation of the global system and is defined in terms of the uses that are to be made of the output of $p_2(s_{p2})$. Even if the output is not exactly the same as that produced by p_1, the output may be sufficient to produce the same overall result of the larger system. Many examples of this arose in Mostow's and Cohen's work. For example, while the original process may have created a copy of one of the inputs as the output, it may be acceptable for the new process to return a pointer to the original input, as long as no destructive operations are performed on the input or output by later processing.

If knowledge compilation is to improve the efficiency of the total system, then using p_2 must be more efficient than p_1 for those situations in S_{p2} to which it is applied. Efficiency must be judged according to a suitable metric, such as the time or space it takes to produce the output. Knowledge compilation may not improve every instance of the process, but there should be a net gain. Mostow and Cohen provide a cost-model for caching that is easily applied to other forms of knowledge compilation [15].

Even when knowledge compilation is successful there can be problems. First, compiling a process that is incorrect will lead to a compiled process that is also incorrect. Another problem can arise when

the original process is improved through some modification. The compiled process will reproduce the original behavior and not reflect the improvement that has been made. This is essentially a cache consistency problem, and to recover from it, the compiled process must either be removed or modified. However, this property of knowledge compilation can also be beneficial when the compilation process saves an instance of original processing that would otherwise be irreproducible because it was based on some temporary external input.

Knowledge Compilation in Soar

In Soar, chunking performs knowledge compilation, as defined above. The processes it compiles are problem-solving episodes in subgoals. The initial situation, s, for a process is defined by the state of working memory before the impasse occurs. The outputs of the process are the results of the subgoal. The actual input to the process consists of those aspects of the pre-impasse situation that were used in creating the result. Chunking creates new rules based on instantiated traces of the production firings that led to the creation of the result. A production trace is only a partial description of a single situation (all of working memory) and the process (all of production memory). It includes traces of productions that contributed to the generation of the problem spaces, states, operators, and results. It does not include traces of productions that influenced the selection of these objects by the decision phase. These selections should only influence the efficiency of the problem solving, not the ability to produce the final results. Knowledge compilation need only be sensitive to those aspects of the original process that were logically necessary, not those that contributed only to the efficiency of the process.

The compiled process is a rule that summarizes the processing in the subgoal. The creation of a rule from the trace information requires many steps which are described in detail elsewhere [4,5,6]. The input to a rule is the set of working-memory elements that match its conditions. A rule's conditions test only a subset of the total situation, thereby allowing the rule to possibly apply in many more situations than in which it was learned. When a rule fires in some future situation, it is able to replace the original processing in the subgoal (by adding elements to working memory that avoid an impasse).

There is no guarantee in the current implementation of Soar that the additional cost of matching the rule will be less than problem solving in the subgoal. Work in other systems has shown this to be problematic [13]. In practice, the rules can be matched efficiently [3, 22] so that the overhead of adding a new rule is much less than solving the subgoal.

Overgeneralization

A compiled process is overgeneral if it applies to situations that are different from the ones on which it was based and does not produce acceptable results. A more formal definition of overgenerality is:

> Process p_2 is overgeneral if there exists situations s^- and s^+, elements of S_{p2}, such that the input of s^- for p_2 is a proper subset of the input of s^+ for p_1 and $p_1(s^+) \neg\sim p_1(s^-) \sim p_2(s^+) \sim p_2(s^-)$, where $\neg\sim$ means not acceptably the same.

A key provision for overgenerality is the subset relation that guarantees that there is a situation (s^+) with inputs that the original process (p_1) is sensitive to, but that the compiled process (p_2) is not. When those aspects of the input are present in a situation, the original process produces different results, $p_1(s^+) \neg\sim p_1(s^-)$, but the compiled process produces the same result, independent of the additional aspects of the situation, $p_2(s^+) \sim p_2(s^-)$. Therefore, although the compiled process performs correctly when the additional aspects are not present, $p_1(s^-) \sim p_2(s^-)$, it does not produce an acceptable result when they are, $p_1(s^+) \neg\sim p_2(s^+)$.

Overgeneralization is a more specific problem than just an incorrect compilation, which would have $p_2(s) \neg\sim p_1(s)$ for some s that is an element of S_{p2}. Overgeneralization also differs from the valid transfer of knowledge to new situations. With transfer, either any additional aspects of the situations not included as input to p_2 are not relevant to the outcome of p_1, or p_2 is able to correctly handle the newly relevant inputs: $p_1(s^+) \sim p_2(s^+)$ and $p_1(s^-) \sim p_2(s^-)$.

Overgeneralization is possible because the compiled process is not restricted to those inputs for which it can produce a correct result. The reason for this failure is that some aspect of the situation that was truly relevant for p_2 to correctly generate its results was not captured in the input to p_2. Any aspect of knowledge compilation can be to blame: the compilation process, the target representation for the compiled process, or the inputs to the compilation process. In the remainder of this section we consider each of these in turn. Our analysis is drawn from examples of overgenerality that have arisen in Soar during the implementation on a wide variety of tasks. As we shall see below, overgenerality is still possible in Soar, but only in special cases that are easily identified.

Invalid compilation.

The first class of failures arises when the knowledge compiler does not create an acceptable process given the information it has available. One cause could be simply a programming problem, the knowledge compiler is just incorrect. However, it might be the case that the complexity of the compilation is such that the resources involved preclude a correct compilation. Partially correct compilations can be used that work in a vast majority of cases, but in those where the complexity is too high, an overly-general compiled process may be produced.

The knowledge compiler can also lead to overgeneralization if it attempts, via an unjustified inductive leap, to generalize the compiled process so that it can be applied to a wider range of situations. In Soar, the only deliberate act of generalization performed by the compilation process is to replace object identifiers with variables. These identifiers are temporary symbols generated for objects as they are inserted into working memory. This is the minimum amount of generalization that is necessary for the learning to have any effect. For each run of the system, these identifiers may have different values, even if the situation is the same. Their exact values are irrelevant, only their equality and inequality is important. Therefore, all instances of the same identifier are replaced by the same variable, and all distinct identifiers are replaced by distinct variables, during the creation of a chunk. This can lead to some overspecialization in that some identifiers are forced to be the same (or different) that did not have to be. However, by itself, it does not lead to overgeneralization.

Insufficient target language

A second source of overgeneralization can be the target language for the compiled process. If it is not possible to represent all of the necessary computation in the target language, the compiled process will be necessarily incomplete, and possibly overgeneral. One aspect of this is that the compiled process must be able to access and test all of the input that was relevant to the original process. For example, when information used in the original process is encoded in different modes, such as symbolically and as activation as in ACT* [1], the compiled process must be able to test all modes. In ACT*, both the original and compiled processes were encoded as sets of rules. The rules could explicitly test for only the existence of working-memory elements, but the interpreter of the rules depended on the activation of working-memory elements to choose between rules. The compiled rules can not encode the dependency of the original process on the activations, so overgenerality is possible. The problem arises from the existence of *crypto-information*, information used by the interpreter that is not explicitly available to the rules [18]. In Soar this problem does not arise because all processing is based only on the pre-impasse working memory, which is available to both the uncompiled, and compiled processes. The production language is rich enough to capture any tests of the contents of working memory that arise from the production firings and decisions that occur in a subgoal.

Insufficient description of process and situations

A final class of failures arise when the process and situation descriptions used by the compiler are insufficient to determine the inputs of the compiled process. In many systems, including Soar, only a partial description of the situation and process being compiled is made available (in Soar's case both are embedded in the production trace). The descriptions of both processes and situations can be decomposed into local features of each component of the process or situation, and global features of the process or situation. Local properties can be determined independently of the other components and thus are usually readily available. In Soar, the local properties are the working-memory elements that

matched the productions that fired on the path to the results. The local features of the process are the contents of these same productions. Global properties, such as whether any working-memory elements do not have a given feature or the reasons why no productions fired during elaboration, are more costly to compute. Knowledge compilation based solely on a trace of behaviour is prone to overgeneralization because traces usually contain only local features of the situation and processing. Of course, if the process being compiled was insensitive to the global properties of its input and itself, overgeneralization will not be a problem. In this section we work our way from local to global features of the situation and then the process. We start with examples from Soar where the necessary information was not provided in early versions but now is. We end with some examples of where global properties of the process are missing so that overgeneralization is still possible.

Goal tests performed by search control. Overgeneralization is possible if the processing within a subgoal that should be irrelevant, influences the results of the subgoal. In Soar this is possible when control information indirectly performs parts of the test for a desired state in a goal. Control information in Soar is encoded as preferences that affect the selection of problem spaces, states and operators. Theoretically, all subgoals should be correctly solvable without any control information, that is, by a search of the problem space until a desired state is achieved. However, often it is possible to simplify the operators of the problem space or the test for a desired state by using control information that explicitly directs the search. The search control is able to guarantee certain invariants so that they need not be tested in the operators or the goal. For example, if the maximum number from a set is desired, the test for the desired state should include comparing a number to all other possibilities (or at least the best found so far). By using control information, those numbers that are less than the one under consideration can be avoided in the search so that the goal is achieved when there are no other numbers to generate. This type of approach would lead to overgeneral chunks because only the processing to generate the best number would be included, not all of the tests that were performed by productions that created preferences.

To eliminate this source of overgeneralization we could have outlawed the use of search-control rules that affected the validity of the result of a subgoal. However, these desired states are often difficult to test explicitly. Instead, we created two new classes of preferences, called required and prohibit. These preferences have special semantics in the decision procedure. Required means that a given object must be selected if the goal is to be achieved. Prohibit meaning that a goal cannot be achieved if a given object is selected. If a required or prohibit preference is used by the decision procedure on the path to a result, the production that generated that preference is considered to be necessary for the generation of the result (as are its predecessors that generated the working-memory elements that it tested). This eliminates the cause of overgeneralization.

Negated tests of the pre-impasse situation. In Soar, a production can test for the absence of an element in working memory. These tests are called negated conditions. Negated conditions allow Soar to realize certain types of default processing [17], which makes the processing in the subgoal defeasible [2]. A defeasible process is one where a different result can be produced when more information is available.

In Soar, additional working-memory elements can prevent a production from firing, thereby leading to a different result. We initially believed that overgenerality was directly related to defeasibilty, however as long as the reason for defeasibility (the absence of elements in working memory for Soar and the absence of certain inconsistencies in a default logic) is included in the compiled process, overgenerality can be avoided [19]. For example, in Soar's original chunking mechanism, the production instantiations only contained those working-memory elements that matched the conditions that tested for the presence of working-memory elements. Since the chunking mechanism did not have access to the negated conditions, the chunks it built had the potential of being overgeneral. To eliminate this problem, Soar was modified so that it included instantiated versions of the negated conditions. Although the negated conditions test global properties of working memory, they are local properties of the productions and easily computed.

The remaining cases of overgeneralization in Soar occur when the processing in a subgoal depends on global properties of the process being compiled, that is, on the contents of all of production memory. The information that chunking is missing consists of traces of some of the productions which did not fire. This is not provided because of the computational burden that would be involved in analyzing it during learning and because we believe that an explicit representation of procedural knowledge (in this case the production memory) may not exist. Two cases of this have been identified, and we discuss them below.

Negated tests of the post-impasse situation. If a production in a subgoal contains a negated condition that test for the absence of working-memory elements that are generated within the subgoal, it is inappropriate to include that negated condition in the chunk built for the subgoal. Instead, if the chunk built for the subgoal is to be completely correct, all productions that might have created that working-memory element in the subgoal, but *didn't*, must be found and the reasons for their inability to fire must be included in the chunk. If we do not allow the examination of production memory, then overgeneral chunks are possible. Even if all productions can be examined, the process to determine the appropriate chunks is expensive, possibly resulting in multiple productions for each result, with very large numbers of conditions for each production.

Tests for quiescence. A second global property of the production system that can influence the processing in a subgoal is that the elaboration phase runs until quiescence, that is, until no more productions can fire. If the productions are able to detect that a specific level of processing was performed, and then no more was possible, they are testing a global property of the productions. One example of this arises in testing that a goal has been achieved. For some goals, detecting that all possible processing was performed without error appears to be a sufficient goal test. For example, unification succeeds when all variables successfully unify. Often the easy test to perform is that there is nothing more to do. This can be detected in Soar because an impasse will arise when all of the operators for a state are exhausted. This impasse can be used as a signal that the subgoal is finished. Testing for the impasse is using information concerning the contents of production memory. If additional information was available in the state, a production might fire and create an additional operator to perform more processing. This leads

to the creation of an overgeneral chunk that will only achieve a part of the goal in a similar situation in which there is more processing to be done. The chunk will only test and handle those aspects that were tested in the original subgoal, leaving some processing undone. To solve this problem by accessing the productions would be similar in complexity to handling the negated conditions that test internally generated working-memory elements because in both cases the chunking mechanism must detect why certain productions did not fire.

Avoiding and Recovering from Overgeneralization

If there is a potential for overgenerality because of insufficient information about the process, overgenerality can be avoided by three obvious approaches: 1. supplying the missing information; 2. eliminating that aspect of the process that is unavailable; 3. refraining from building chunks for subgoals where information is missing. For two cases we described earlier (negated tests of the pre-impasse situation, and goal tests performed by search control), we have modified Soar to provide more information, thereby eliminating that aspect of the process that is unavailable. For the two outstanding causes of overgeneralization (negated tests of the post-impasse situation, and tests for quiescence), we balk at supplying the information, first because we expect situations in which the contents of production memory are unavailable, and secondly, because of the computational cost of analyzing that information. One option is to eliminate the use of negated conditions for internal objects, and eliminate the ability to base behavior in a subgoal on the creation of an impasse. If we adopt this approach, we must first find ways of replacing their functionality. Another alternative is to avoid chunking subgoals in which there is processing that could lead to overgeneralization. The chunking mechanism is able to detect tests for internal negated elements and tests for impasses that arise from exhaustion. Disabling chunking for these cases appears to be a prudent step for now.

Another alternative to avoiding overgenerality is to modify the subsequent processing so that the definition of acceptably-the-same is more lenient. This can be achieved by moving some of the test for a result of a goal outside the goal so that the goal produces results that only potentially satisfy the goal. The test does not get included in the chunk, but always follows the application of the chunk, screening the results. The chunks that are learned then can be overgeneral, but the following test will not use them to eliminate the impasse unless they are appropriate. One restriction is that it must be possible to recognize a acceptable result with a simple test. This type of scheme has been implemented in Soar as part of work on abstraction planning [24], for a part of R1-Soar [21], a reimplementation of R1 [11]. In a subgoal to evaluate one of the operators in R1-Soar, the operator is implemented at an abstract level, so many of the details are left out. A chunk learned for that implementation applies later when a complete implementation is required. The chunk partially implements the operator, but does not pass an additional test that is needed before the result it used. Therefore, an impasse arises, followed by problem solving that completes those aspects of the operator left out of the chunk.

Even without an additional test, if the purpose of the subgoal is to select between operators, an

overgeneral chunk will not lead to incorrect results, although it may lead to additional search. In addition, the level above will be correctly chunked because all search-control knowledge is ignored during chunking.

A third way to avoid overgenerality is to have the knowledge-compilation system modify the compiled process in anticipation of overgeneralization. This is done to a small extent in Soar for a specific type of overgeneralization that arises when operator-implementation subgoals are chunked. Since working-memory elements can not be changed (only added) operators are usually implemented by creating a new state, copying over all unchanged information, and adding all new information. If this is performed in a subgoal, the copying is usually handled by copying all subparts of the state that are not marked as being changed. This is a case of processing until exhaustion because the copying is independent of the number of subparts. This will lead to an overgeneral chunk that applies whenever the appropriate operator is selected and there is a state that has at least those subparts that were modified or copied in the original subgoal. If there are additional subparts to be copied, the chunk will apply, not just once, but for every appropriately sized subset of subparts, creating multiple states, none of which is complete. To avoid this problem, the chunking mechanism analyzes the rules that it is building, and if they appear to be copying substructures, splits the rule into multiple rules so that the copying is carried on independently. This then allows the copy rules to fire to exhaustion, copying all subparts as necessary. Although this has always worked in practice, the determination that the chunk should be split into multiple rules is only heuristic This undesirable interaction between operator implementation and overgenerality suggests that a different scheme for applying operators may be needed.

An alternate approach is to explicitly modify the overgeneral process either by further restricting its inputs, either by modifying its processing or removing it. In a rule-based system like Soar this would mean adding new conditions to an overgeneral ruleor deleting the rule from production memory [8,23]. In other rule-based systems that fire only some of all the matched rules , additional properties of the rule can be modified. In ACT*, the *strength* of a rule can influence its ability to fire. When a rule is suspected of being overgeneral or incorrect, it strength is lowered [1]. We have rejected schemes that modify existing productions, preferring, for the moment, to investigate ways of learning new productions that overcome the overgeneral ones.

Summary

In this paper we have attempted to categorize most if not all of the sources of overgeneralization that can arise during knowledge compilation. We've used this categorization to analyze a form of knowledge compilation, chunking in Soar. Of all the sources of overgeneralization, one is inherent to our architectural assumption that the knowledge-compilation process does not have access to all of long-term knowledge, which in Soar's case is its production memory. We've identified two types of processing in Soar that can give rise to overgeneralization if this information is not available and we have proposed ways of avoiding the creation of the overgeneral chunks and then recovering from them if avoidance is not possible.

References

[1] Anderson, J. R. *The Architecture of Cognition.* Cambridge: Harvard University Press, 1983.

[2] Batali, J. Computational Introspection. A. I. Memo 701, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February, 1983.

[3] Forgy, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence19,* (1982),17-37.

[4] Laird, J. E. Soar User's Manual (Version 4). ISL-15, Xerox Palo Alto Research Center, 1986.

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

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

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

[8] Langley, P. , & Sage, S. Conceptual clustering as discrimination learning. Proceedings of the Fifth Conference of the Canadian Society for Computational Studies of Intelligence, 1984.

[9] Lewis, C. H. *Production system models of pratice effects.* Ph.D. Th., University of Michigan, 1978.

[10] Marsh, D. Memo functions, the graph traverser, and a simple control situation. In B. Meltzer & D. Michie (Ed.), *Machine Intelligence 5.* New York: American Elsevier, 1970.

[11] McDermott, J. "R1: A rule-based configurer of computer systems". *Artificial Intelligence , 19* (1982) 39-88.

[12] Michie, D. "Memo" functions and machine learning. *Nature,* 1968, *218,* 19-22.

[13] Minton, S. Selectively generalizing plans for problem-solving. Proceedings of IJCAI-85, Los Angeles, 1985.

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

[15] Mostow, J. & Cohen, D. Automating program speedup by deciding what to cache. Proceedings of IJCAI-85, Los Angeles, 1985.

[16] Neves, D. M. & Anderson, J. R. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In Anderson, J. R. (Ed.), *Cognitive Skills and their Acquisition.* Hillsdale, NJ: Erlbaum, 1981.

[17] Reiter, R. "A logic of default reasoning". *Artificial Intelligence13* (1980), 81-132.

[18] Rosenbloom, P. S. *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility,* Ph.D. Th. Carnegie-Mellon University, 1983.

[19] Rosenbloom, P. S., & Laird, J. E. Mapping explanation-based generalization onto Soar. Proceedings of AAAI-86, Philadelphia, 1986.

[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., Newell, A. & 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] Scales, D. Efficient matching algorithms for Soar/Ops5 production systems. Computer Science Tech. Report, Stanford University, 1986.

[23] Sonnenwald, D. H. Over-generalization in Soar., In preparation.

[24] Unruh, A., Rosenbloom, P. S., & Laird, J. E. Implicit procedural abstraction. In preparation.

Mapping Explanation-Based Generalization onto Soar[1]

P. S. Rosenbloom, Stanford University, and J. E. Laird, Xerox Palo Alto Research Center

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]

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

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.