On the Operationality/Generality Trade-Off in Explanation-Based Learning*

Alberto Maria Segre†

87-830

April 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

---

*This paper also appears in the *Proceedings of the Tenth International Joint Conference of Artificial Intelligence,* Milano, Italy (1987).

†Support for this research was provided by a Caterpillar Corporation Graduate Fellowship, the Air Force Office of Scientific Research under grant F49620-82-K-0009, and the National Science Foundation under grants NSF-IST-83-17889 and NSF-IST-85-11542.

On the Operationality/Generality Trade-Off in Explanation-Based Learning*

Alberto Maria Segre†
segre@gvax.cs.cornell.edu

Abstract

In this paper we examine the *operationality/generality trade-off* and how it affects performance of *explanation-based learning* systems. Experience with the ARMS learning apprentice system, presented in the form of an empirical performance analysis, illustrates both sides of the trade-off.

1. Introduction

Recent work in *explanation-based learning* (EBL) has attracted much attention from the machine learning community [DeJong86,Mitchell86]. A number of systems have been and are still being developed in a spectrum of domains from natural language processing [Mooney85], to circuit design [Mitchell85a].

This paper discusses the *operationality/generality trade-off* and how it arises in EBL in general and the implementation of the ARMS (for **A**cquiring **R**obotic **M**anufacturing **S**chemata) system in particular [Segre85,Segre87a,Segre87b]. ARMS is a *learning apprentice system* [Mitchell85b] that learns to assemble simple mechanical devices using an idealized robot arm. By observing, analyzing, and generalizing a human planner's solution to an assembly episode, ARMS builds a new schema which can be used to synthesize assembly plans for future episodes.

We begin with an intuitive discussion of the operationality versus generality problem. We then proceed to give a short description of the ARMS system [Segre87b] with emphasis on the generalization process. Finally, we describe a simple ARMS example to illustrate this important trade-off.

2. Operationality versus Generality

Explanation-based learning (EBL) systems are capable of acquiring knowledge from a single example. Using a domain theory, a sample problem solution is analyzed in order to account for how the goal is accomplished. This analysis, or *explanation*, is generalized to create a new knowledge structure.

If the newly acquired knowledge structure is available to the system, it is said to be *operational*. If the new structure does not improve the performance of the system in some fashion, it is not worth learning. However, not all operational knowledge is created equal: the cost of using the new knowledge structure is called its *operationality*. The more operational the structure, the easier (e.g., less expensive) it is for the system to apply it.

Note that we have yet to address the issue of exactly *how* the new structure is used. The only important point is that the knowledge should increase some aspect of the system's capability or performance. The new knowledge structure might be used in order to construct explanations of more complex examples, thereby increasing a system's comprehension ability. On the other hand, it might be used by a system's performance element (e.g., a

---

* This paper also appears in the *Proceedings of the Tenth International Joint Conference of Artificial Intelligence*, Milano, Italy (1987).

† Support for this research was provided by a Caterpillar Corporation Graduate Fellowship, the Air Force Office of Scientific Research under grant F49620-82-K-0009, and the National Science Foundation under grants NSF-IST-83-17889 and NSF-IST-85-11542.

planner) in similar problem-solving situations.

But just how similar is similar? The diversity of examples covered by the new structure is directly related to its *generality*: the more general the knowledge, the more situations where it is likely to be useful. Unfortunately, the more general the structure the more expensive its application tends to be. We term this dichotomy the *operationality/generality trade-off*: a more general structure is less operational and vice versa.

Consider an example form the game of chess. Having observed the successful capture of the opponent's rook using the technique called a *knight fork*, a system could produce a very operational new structure describing *exactly* this board situation.[1] For the system to use this new structure in a future game, every piece on the chess board would have to occupy exactly the same position as in the first example.

Clearly this new structure, while operational to an extreme, is hardly general. The position of a pawn on the other side of the board can hardly affect the success of the knight fork. On the other hand, it is quite simple and inexpensive to use: in the same board situation, application of the structure is immediate and practically without cost.

If, however, the rook capture is analyzed and generalized, the resulting knight fork knowledge structure might well take into account the relative positions of the knight, the decoy, and the victim (a rook in the observed example). It should not rely on the actual positions of these pieces, since the knight fork is applicable anyplace on the chess board. In addition, the piece type is only important for the knight: the captured piece need not always be a rook.

Such a new structure is more general, since it can be applied to many other board situations. Naturally, the cost of applying this new structure is greater than the trivial case described above. First, we must decide whether the structure is applicable to the current situation. This involves more than a simple matching, since our structure does not fully specify the chess board. Second, we must identify which pieces on the board play which roles in the structure.

Designers of EBL systems must decide where the new structure produced by their system's generalizer lies in this operationality/generality spectrum. Mitchell *et al* [Mitchell86] posit a fixed *operationality criterion* to specify the level of representation used for the new knowledge structure. They suggest that a fixed vocabulary be established *a priori*, so that any new structure described using this limited vocabulary is by definition considered operational.

As pointed out by Mooney and DeJong [DeJong86], there are obvious difficulties involved in finding such a fixed operational vocabulary for any given domain. Even if a fixed operationality vocabulary could be specified, there is no guarantee that structures expressed with this vocabulary are easy to apply. As an example, Mooney and Dejong give the following two propositions:

PROVABLE("2+2=4")
PROVABLE("FERMAT'S LAST THEOREM")

The first is clearly operational, while the second is not. Yet both are described using the same vocabulary.

In any case, the fixed vocabulary approach seems needlessly limiting, especially if one departs from the static classification type tasks of Mitchell *et al* and considers acquiring problem-solving skills. When learning problem-solving skills, the system is acquiring the

---

[1] This trivial case of learning is normally called *memorization* or *rote learning*. The careful reader will notice that rote learning does not really qualify as EBL since there is no generalization step. It does, however, serve to illustrate operationality taken to the limit.

ability to plan solutions (sequences of operators) to other problems by analyzing a trace of a successful solution.

Mooney and DeJong present an alternative, more dynamic, definition of operationality based on the problem-solving capabilities of a system. In their view, an EBL system makes previously known static concepts operational by learning how to *achieve* these concepts. For example, the concept of *checkmate* in the game of chess is easy to describe (a situation where one player's king cannot escape capture) but difficult to operationalize (planning a series of operators to achieve checkmate). A concept therefore becomes operational only when the system acquires and can index a plan to achieve it.

Making a plan operational makes no statement about the quality of the plan. From our earlier discussion, it is clear that there are many different levels of operationality, levels that will affect system performance. In the next section, we describe an example from the ARMS system which compares two different levels of operationality.

3. The ARMS System

We now turn our attention to an example taken from the ARMS learning-apprentice system [Segre85,Segre87a,Segre87b]. ARMS is an EBL system that acquires the ability to plan sequences of robot motions to accomplish assembly of simple mechanisms. A complete description of the system is far beyond the scope of this paper, however, a short overview describing the general structure of the system will be helpful when describing the example.

ARMS learns by unobtrusively observing an expert guide the robot through an assembly task via the robot arm's teach pendant. To the user, this method is indistinguishable from the *teach-by-guiding* robot retraining method used in most current robot arm installations. In contrast to teach-by-guiding systems which provide for rote memorization of the input sequence, ARMS acquires, from a single episode, the power to plan the assembly of an entire class of functionally similar mechanisms.

The ARMS architecture is that shown in Figure 1. It divides into two elements, the *learning element* and the *performance element,* which do not operate concurrently. The two elements share domain knowledge which is stored in the form of *schemata* in the *schema library.*

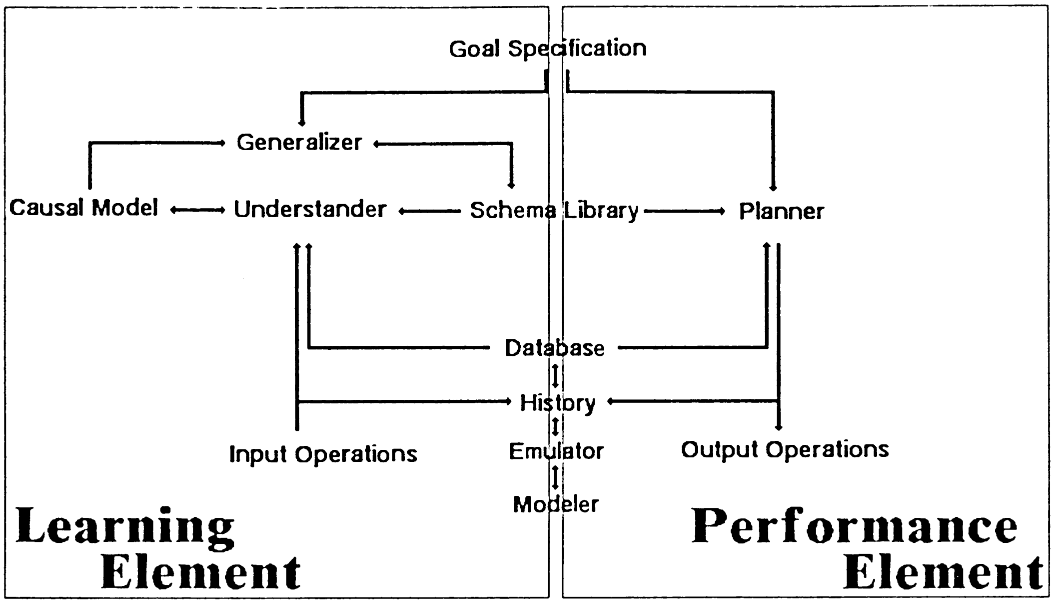
The user specifies an assembly by giving a *functional goal specification.* The goal specification describes the mechanical behavior of the desired assembly, without specifying a physical description. If the performance element can derive a physical goal description from the functional goal specification (the *design problem*), it then attempts to derive a plan to transform the initial world state into a state containing an instance of the design-problem solution (the *assembly problem*).

If the performance element fails to find solutions to the design and assembly problems, control is transferred back to the user, who now guides the learning element through the assembly process. As the user moves the arm about the workspace, the system observes the user's solution and produces a new schema to use in future problem-solving situations.

3.1. World Model

The ARMS system relies on an emulation of the robot world in order to reason about how pieces fit and move together. This emulator is much like the modeling systems used for computer graphics and CAD/CAM applications. It is a simple *constructive solid geometry* (CSG) modeler which represents pieces as the sum and/or difference of primitive volumes.

The modeler is used to model the ARMS gripper and its interactions with the world.[2] It must realize when the gripper is manipulating a piece, what effects these manipulations have on the position of that piece and any piece interactions. The modeler maintains a copy of the world at each time tick by using a storage-efficient mechanism similar to those used



ARMS System Architecture

Figure 1

for copying sparse matrices.

The rest of the ARMS system does not have direct access to the emulation afforded by the CSG world modeler: rather it only has access to descriptions of this world which are maintained by the database. Any request for world information is shuttled through the database, which manipulates tokens representing relations between elements of the world model. This collection of tokens corresponds to the state of the problem-solving domain. Note that it is the responsibility of the database to uniquify requests so that there is only one copy of a symbolic state to describe a particular partial world state.

3.2. Knowledge Representation

All domain knowledge in ARMS is represented as *schemata*. A schema [Chafe75,Charniak78,Minsky75,Schank77] is a general structure which, via the use of *slots* which can be filled with a variety of different values, can be used to represent a class of similar concepts. A schema is said to be *instantiated* when all of its slots are bound to constants or other instantiated schemata, in which case it represents a unique concept.

ARMS schemata fall into five different categories:

(1) *Physical object schemata* represent the CSG models of the pieces in the domain. While certain limitations are imposed by the simplicity of the modeler, ARMS can model an infinite set of different pieces.

(2) *State schemata* represent the vocabulary with which ARMS reasons about the real world. These are maintained by the database in an on-demand fashion: no state schema is ever generated or otherwise manipulated except as a result of an explicit request by the system. Each state contains temporal information indicating start and

[2] While ARMS has been used to drive a real robot arm [Gustafson86], its normal mode of operation involves driving the simulated robot arm in this simulated environment.

end times for the state.[3]

(3) *Constraint schemata* are just like state schemata, except that they are time-invariant. Certain aspects of the world are not mutable by applying system operators (e.g., relations between sizes of pieces): thus the constraint schemata are maintained more efficiently by the database. Constraint schemata are attached to state schemata to impose restrictions on the values the state schema slots may take.

(4) *Joint schemata* are also like state schemata in that they are maintained by the database and have temporal scope. They come in two flavors: *abstract joint schemata* describing the mechanical behavior of two related pieces, and *physical joint schemata* describing the implementation of the mechanism in terms of interacting surfaces. Together, these joint schemata form the building blocks of the ARMS domain theory.[4]

(5) *Operator schemata* (which include the five *primitive operator schemata*) represent the plans the system can apply. The primitive operator schemata correspond to the idealized ARMS robot arm command set. They provide the lowest level of description for robot motion. Operator schemata describe the context in which they can be applied, as well as what goals they achieve. The composite (e.g., non-primitive) operators are recursively defined in terms of other schemata. The system learns this kind of schema, and it is this case we will examine a bit more closely later in our operationality/generality discussion.

3.3. The Performance Element

Inputs to the performance element are:

1. The initial state of the world;
2. A goal specification given as an abstract joint schema.

The performance element produces a sequence of fully instantiated primitive operator schemata.

The ARMS performance element begins with a *design phase*, where a physical joint schema consistent with the functional goal specification (an abstract joint schema) is derived. This physical joint schema is then used to index into the schema library and select a top-level plan. The *planning phase* recursively expands the top-level plan to produce a robot arm command sequence which achieves the final state from the specified initial state.

The planning process is a depth-first search through the plan space defined by the operator schemata (both built-in and acquired) stored in the schema library. This *schema planner* (similar to the skeletal planner of [Friedland79]) is a simple design which selects an abstract plan (in the form of an operator schema) to achieve the specified goal state, and repeatedly expands it until the process bottoms out with a robot arm command sequence.[5]

---

[3] Even though state schemas are checked against the emulator only as a result of a request, the amount of time spent by the database in satisfying requests for symbolic information about the world accounts for a large portion (in some examples as much as 96% of the total) of the computational resources expended by the ARMS system. While ARMS runs on a serial machine, this kind of database mechanism is a prime example of those algorithms which seem best suited to large grain size parallel machines.

[4] The system is capable of learning physical joint schemata, but the finite set of abstract joint schemata is built in (the reader is referred to [Segre87b] for a discussion).

[5] The ARMS planner stops at the level of primitive robot arm commands. When driving the robot arm, the primitive operator schemata are expanded into arm-dependent control statements: this is done with an arm-specific algorithm which takes robot level statements down to the kinematic level. Note that ARMS does not deal with collision avoidance, an active research topic which goes well beyond the scope of this implementation. For now, ARMS assumes an uncluttered workspace so that collision avoidance is not a problem.

3.4. The Learning Element

The learning element is responsible for first understanding how the user solved the problem, and then generalizing the observed solution into something that can be used again later by the performance element. We divide this task into three distinct subtasks: *understanding*[6], *verifying*, and *generalizing*.

Inputs to the learning element are:

1. The initial state of the world;
2. A goal specification given as an abstract joint schema;
3. A sequence of instantiated primitive operator schemata.

The primitive operator schemata are echoed from the teach pendant of the robot arm being led through the assembly episode by the user. The understander builds a causal model of the external agent's problem solving behavior.

When the user is finished, the system *verifies* that the function of the physical mechanism constructed by the user corresponds to the initial functional goal specification. The verification process relies on a naive kinematic *domain theory* to analyze the function of a mechanism. This process may result in the acquisition of a new schema via *explanation-based specialization* that operationalizes the functional goal specification and provides a solution to future design problems. (see [Segre87b] for a more thorough description of the verifier).

Next the *generalizer* produces another schema via *explanation-based generalization*. This newly acquired schema can then be used to solve an entire class of assembly problems which share the same functional goal specification.

In this paper, we give a quick description of the generalizer only: the reader is again referred to [Segre87b] for a more thorough discussion.

3.4.1. The Generalization Process

The *generalizer* takes as its input the verified goal specification given by the user and the causal model produced by the understander. The generalizer produces a new composite operator schema which can be used both in understanding and planning (see Figure 2).

As a result of the verification process, an instantiated version of the user-specified abstract joint schema is tied to a set of physical joint schema tokens in the causal model. These tokens constitute the *top-level subgoal set.*

We begin by ordering the top-level subgoal set on the basis of a causal dependency analysis. This causal analysis relies on the ARMS domain theory to determine if there are any ordering dependencies between elements. In short, the limits of travel in the individual subjoints of the mechanism are examined to see if their boundary conditions rely on other subjoints.

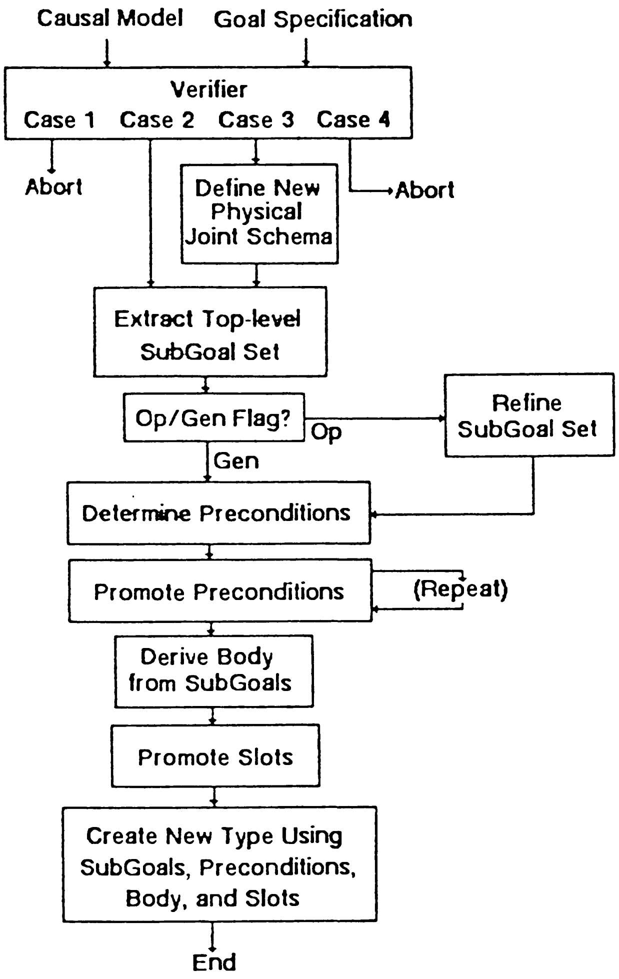
To extract the explanation from the causal model, it is sufficient to follow the pointers established during the understanding process from the (now causally ordered) top-level subgoal set down to the primitive operator inputs. The relevant pointers are those connecting operators to their recursive expansions. Note that an explanation should also contain pointers to all of the constraint schemata supporting states in the explanation.

---

[6] Apologies to McDermott:

> We should avoid, for example, labeling any part of our programs an "understander." It is the job of the text accompanying the program to examine carefully how much understanding is present, how it got there, and what its limits are [McDermott76].

While we often use the alternate, but less illustrative, term *justification analyzer*, which is perhaps more acceptable from McDermott's point of view, this practice conflicts with our goal of descriptive simplicity.



Generalization Flowchart

Figure 2

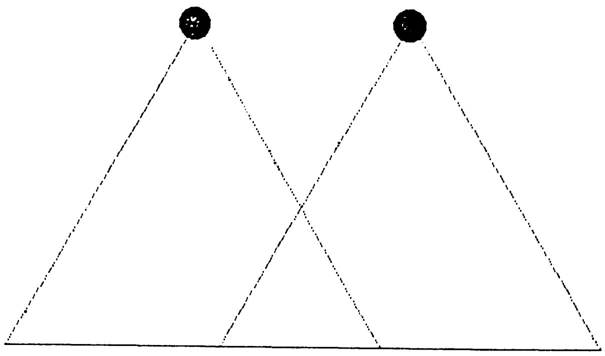
At this point, the generalizer must pick a level of representation for the new operator schema. The higher the level of representation, the more generally applicable the new schema will be: however, a price will be paid in the amount of work done by the planner in applying the new schema. The extra work comes from the added levels necessary in the recursive expansion of the plan. Conversely, a more operational new schema will be easier to apply, but less generally applicable.

Depending on the value given by the user to the operationality/generality parameter, the ARMS generalizer expresses a new schema as follows:

(1) as the abstraction of the top-level subgoal set, producing a more general new schema (see Figure 3).

(2) by descending the explanation structure to a level where all of the state schemata are roots of independent subtrees in the explanation (a more operational new schema). The state schemata at this lower level become the subgoal set of the new composite operator schema. This produces a new subgoal set which preserves any causal dependency orderings established during the joint analysis (see Figure 4).

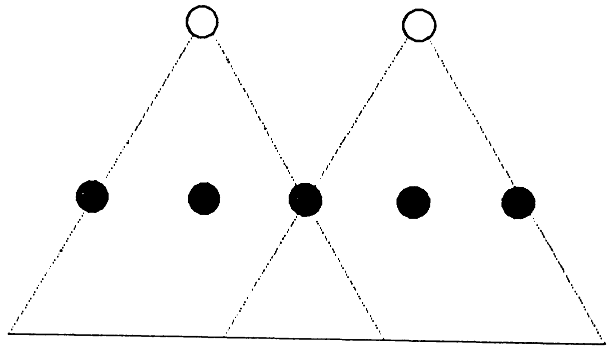
Having in this fashion collected a set of subgoals, we now complete the construction of the new operator schema and integrate it into the schema library.



Level with Shared Substructures in Explanation

The two elements of the top-level subgoal set in this example are represented as black nodes. Their respective explanation substructures are outlined as triangular subtrees. Shared substructure is represented by the overlapping sections of the subtrees. When producing the more general new schema, the generalizer uses the black nodes of the top-level subgoal set as the subgoal set model for the new schema.

Figure 3



Level with No Shared Substructures in Explanation

The two elements of the top-level subgoal set are represented as white nodes at the root position of two overlapping explanation subtrees. When producing the more operational new schema, the generalizer descends into the explanation structure until it can produce a subgoal set (represented here as black nodes) with no shared substructure. This set then becomes the subgoal set model for the new schema.

Figure 4

At first glance, the two schemata do not seem terribly different. In fact, both new schemata are equally capable of solving a set of similar problems, yielding identical solutions. While their operationality difference is obvious in the levels of recursive expansion required in planning, their generality difference is not immediately evident.

Closer examination shows that the more general new schema is better able to integrate subsequently acquired schemata in its planning behavior. In other words, as the system learns how to do things in other ways, the more general new schema, unlike the more operational new schema, is immediately able to take advantage of this new knowledge.

4. An Example

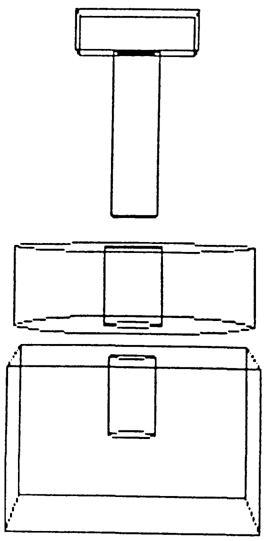
Consider as an example a mechanism (see Figure 5) assembled from three pieces: a washer, a bored block, and a peg. The shaft of the peg is inserted first through the hole in the washer and then into the hole in the block. The washer spins freely about the peg, while the peg fits snugly into the bored block. We call this simple mechanism a *widget.*

We can describe the widget physically by describing the pieces (call them \$Peg1, \$Washer1, and \$BoredBlock1) and the *mating conditions* between them (this is the approach taken by most task level programming systems such as RAPT [Popplestone80]). We could also describe this mechanism at a functional level as a *revolute joint*, e.g., a single rotational degree of freedom, between \$Washer1 and \$BoredBlock1. Unlike the physical description, the functional description need not explicitly mention \$Peg1. In fact, the same functional description would hold for any assembly having the requisite one rotational degree of freedom, regardless of the physical mechanism used to achieve this function.

The system is first given the piece descriptions and their initial placements in the workspace (see Figure 6 for an example). When asked to achieve a revolute joint between the washer and the bored block, the system should automatically generate a sequence of fully instantiated (i.e., with all parameters bound) primitive operators which, when applied by the robot arm, transform the initial state into the assembly described above. The sequence should take into account factors such as the proper grasping strategy for each piece, whether pieces are cleared off before attempting to move them, and so on.

Since the system currently has no plan to achieve this concept (in fact, the system does not yet possess a concept corresponding to this function's physical realization), the system admits defeat, and the user proceeds to show it how to build a mechanism which has the desired functionality. The user guides the system through an assembly episode which results in a physical assembly that fits the specified goal. Note that the assembly sequence given by the user (input to the system as fully instantiated primitive operators) need not be an optimal sequence, but simply effective in accomplishing a physical instantiation of the functionally specified goal.

The system verifies, using its domain theory, how the physical assembly realized instantiates the desired functional goal. During the verification process, a new concept corresponding to this physical realization of the functionally specified goal is acquired.



The Widget

Figure 5

If the verifier terminates successfully, the generalizer produces either a more general or a more operational new schema which can be used to solve similar problems.

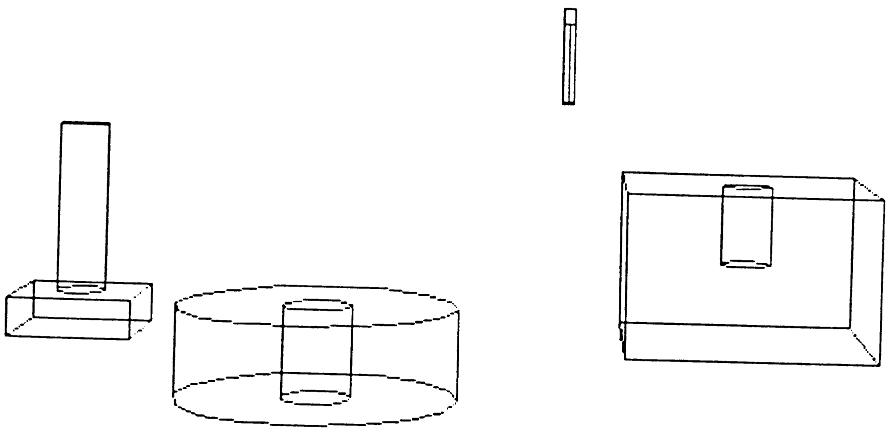
5. Results

In this section, we present some empirical results collected from the ARMS system. ARMS is implemented using the object-oriented language LOOPS, which is in turn constructed in INTERLISP-D. These results were collected on a Xerox 1109 Lisp Machine running the Koto release of INTERLISP-D and the Buttress version of LOOPS. The 1109 has 3.5 megabytes of main memory, a 43 megabyte hard disk drive, and a hardware floating point coprocessor.

Performance of the system is adversely affected during these tests by the information-collecting mechanism. The system suffers a factor of eight slowdown while collecting these statistics. However, since our only interest here is in comparing one example with the other, the slowdown effects are not relevant.

5.1. Learning Episode 1

Given the initial configuration shown in Figure 6, the system is presented with a sequence of 12 primitive operator schemata which complete the assembly of the widget shown in Figure 3. The system constructs a new, more operational, schema which can plan the assembly of this and other functionally similar mechanisms.



Initial State 1

The robot gripper is located in the center of the picture with fingers closed and pointed down. $BoredBlock1 is to the right, $Peg1 is to the left, and $Washer1 is in the foreground just left of center. The functional goal specification is given as an abstract joint schema $RevoluteJoint[$BoredBlock1, $Washer1].

Figure 6

Table 1 Learning Episode 1	
Length of observed sequence	12
Causal model size (tokens)	159
Explanation size (tokens)	144
Number of database queries	1129
Number of tokens created	2090
Number of requests issued	110913
Number of slot reads	103892
Number of slot writes	4581
Total CPU time (seconds)	2039.1
Generalizer time (seconds)	40.9

The size of the causal model and the explanation are given in terms of tokens, where each token represents a schema instance. This is a pretty good measure of the difficulty of the example: the larger the number, the more complex the analysis or the plan.

The number of database queries, tokens created, requests issued and slot manipulations are a good general indicator of how much work is performed by the system.

The total CPU time reflects the time for emulating, understanding, verifying, and generalizing the example. Also provided is the CPU time for generalization alone.

5.2. Learning Episode 2

This example is identical to Learning Episode 1, except that the more general new schema is constructed.

Table 2 Learning Episode 2	
Length of observed sequence	12
Causal model size (tokens)	159
Explanation size (tokens)	144
Number of database queries	1117
Number of tokens created	2072
Number of requests issued	109639
Number of slot reads	102380
Number of slot writes	4573
Total CPU time (seconds)	2047.8
Generalizer time (seconds)	26.4

As expected, the results shown here are almost identical to the results of the previous, identical, episode. The only difference is in the time spent on generalization. The extra analysis required to produce the more operational new schema is clearly evident in the greater CPU time for generalization in the operational case. This is consistent with expected behavior.

5.3. Problem-Solving Episode 1

We again present the system with the initial configuration of Figure 6. The system is asked to produce a revolute joint between \$BoredBlock1 and \$Washer1. The system solves the design problem and then applies the more operational version of the new schema to the assembly problem, generating a 12 step solution.

Table 3 Problem-Solving Episode 1	
Length of planned sequence	12
Planner search tree size (tokens)	146
Planner subtree size (tokens)	95
Number of database queries	1003
Number of tokens created	1554
Number of requests issued	84754
Number of slot reads	92593
Number of slot writes	2987
Total CPU time (seconds)	1724.5

5.4. Problem-Solving Episode 2

This example is identical to Problem-Solving Episode 1 (see Figure 6), except that the new schema being applied is the more general version. We therefore expect this example to be less efficient, since the planner must work harder when applying a more general schema.

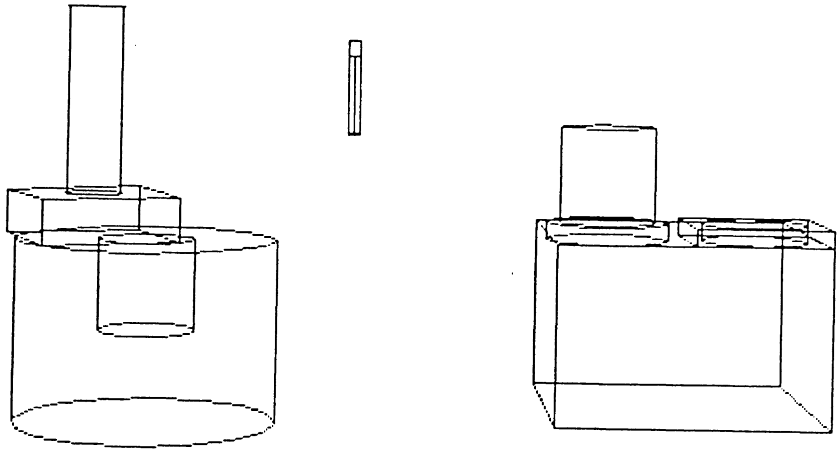
Table 4 Problem-Solving Episode 2	
Length of planned sequence	12
Planner search tree size (tokens)	151
Planner subtree size (tokens)	138
Number of database queries	1340
Number of tokens created	2044
Number of requests issued	118962
Number of slot reads	139695
Number of slot writes	3661
Total CPU time (seconds)	2498.8

The system generates the same solution as in Problem-Solving Episode 1. The solution, however, is more expensive to generate as indicated by the number of tokens generated and requests issued This behavior is also evident in the total CPU time figure (42 minutes as compared to 29 minutes in the other case). In addition, the planner subtree is a bit larger: but since the additional nodes tend to be at the highest level of abstraction, the increase in CPU time tends to be more than linear in the increased subtree size.

5.5. Problem-Solving Episode 3

This problem-solving episode demonstrates the power of the system in planning the assembly of physically different yet functionally similar mechanisms. The system is asked to plan for a revolute joint between $BoredCylinder1 and $Washer2. There is no mention of $Peg3: the system must decide for itself which of the other pieces in the workspace can be used to achieve a physical instance of the functionally specified goal. The initial state is shown in Figure 7.

In this example, we are asking the system to deal with not only a more complicated initial starting configuration, but also the system must construct a functionally specified assembly from physically different pieces.



Initial State 2

The robot gripper is located in the center of the picture with fingers closed and pointed down. $BoredCylinderl is to the left, with $Pegl stacked on top of it. $Peg3 and $Washer2 are stacked (from left to right) on top of $Blockl on the right side of the workspace. The functional goal specification is $RevoluteJoint[$BoredCylinderl, $Waser2].

Figure 7

Table 5
Problem-Solving Episode 3

Length of planned sequence	18
Planner search tree size (tokens)	204
Planner subtree size (tokens)	133
Number of database queries	1753
Number of tokens created	3128
Number of requests issued	174732
Number of slot reads	184900
Number of slot writes	5433
Total CPU time (seconds)	3533.3

5.6. Problem-Solving Episode 4

Problem-Solving Episode 4 is identical to Problem-Solving Episode 3 (see Figure 7). The system supplies the identical solution, but at greater computational expense. This is consistent with expected behavior.

Table 6 Problem-Solving Episode 4	
Length of planned sequence	18
Planner search tree size (tokens)	215
Planner subtree size (tokens)	194
Number of database queries	2357
Number of tokens created	3952
Number of requests issued	256358
Number of slot reads	287017
Number of slot writes	3427
Total CPU time (seconds)	5257.7

6. Conclusion

In this paper, we have provided an overview of the operationality/generality trade-off for explanation-based learning systems. We have described one approach, that of the ARMS learning-apprentice system, for characterizing the operationality of a new schema based on the structure of the explanation itself. Finally, we described an experiment that provides preliminary empirical evidence of the importance of this trade-off.

What we are really dealing with is a set of plans covering a spectrum of operationality/generality characteristics. It is up to the system designer to decide what the best level(s) of operationality are for a given system. Given that there is no single solution, the best we can hope for is to understand the problem well enough to decide where on the continuum a given system's generalizer should reside.

Acknowledgements

The ARMS system was designed and implemented by the author as part of his Ph.D. research at the University of Illinois at Urbana-Champaign. The author is indebted to Professor G. DeJong and the rest of the Coordinated Science Laboratory Artificial Intelligence Research Group or their comments and suggestions.

References

[Chafe75] W. Chafe, "Some Thoughts on Schemata", *Theoretical Issues in Natural Language Processing 1*, Cambridge, MA, 1975, 89-91.

[Charniak78] E. Charniak, "With a Spoon in Hand this Must be the Eating Frame", *Theoretical Issues in Natural Language Processing 2*, Urbana, IL, 1978, 187-193.

[DeJong86] G. F. DeJong and R. J. Mooney, "Explanation-Based Learning: An Alternative View", *Machine Learning 1*, 2 (April 1986). Also appears as Technical Report UILU-ENG-86-2208, AIRG, CSL, University of Illinois at Urbana-Champaign.

[Friedland79] P. E. Friedland, "Knowledge-based Experiment Design in Molecular Genetics", 79-771, Computer Science Department, Stanford University, Palo Alto, CA, 1979.

[Gustafson86] B. Gustafson, "Development of Localized Planner for Artificial Intelligence-Based Robot Task Planning System", M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, October 1986.

[McDermott76] D. McDermott, "Artificial Intelligence Meets Natural Stupidity", *SIGART Newsletter 57* (April 1976), 4-9.

[Minsky75] M. L. Minsky, "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, P. H. Winston (editor), McGraw-Hill, New York, NY, 1975, 211-277.

[Mitchell85a] T. M. Mitchell, S. Mahadevan and L. I. Steinberg, "LEAP: A Learning Apprentice for VLSI Design", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985, 573-580.

[Mitchell85b] T. Mitchell, S. Mahadevan and L. Steinberg, "A Learning Apprentice System for VLSI Design", *Proceedings of the 1985 International Machine Learning Workshop*, Skytop, PA, June 1985, 123-125.

[Mitchell86] T. M. Mitchell, R. Keller and S. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View", *Machine Learning 1*, 1 (January 1986), 47-80.

[Mooney85] R. J. Mooney and G. F. DeJong, "Learning Schemata for Natural Language Processing", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Urbana, IL., August 1985, 681-687. Also appears as Working Paper 67, AIRG, CSL, University of Illinois at Urbana-Champaign.

[Popplestone80] R. Popplestone, A. Ambler and I. Bellos, "An Interpreter for a Language for Describing Assemblies", *Artificial Intelligence 14* (1980), 79-107.

[Schank77] R. C. Schank and R. P. Abelson, *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures*, Lawrence Erlbaum and Associates, Hillsdale, NJ, 1977.

[Segre85] A. M. Segre and G. F. DeJong, "Explanation Based Manipulator Learning: Acquisition of Planning Ability Through Observation", *Proceedings of the IEEE International Conference on Robotics and Automation*, Urbana, IL., March 1985, 555-560. Also appears as Working Paper 62, AIRG, CSL, University of Illinois at Urbana-Champaign.

[Segre87a] A. M. Segre, "A Learning Apprentice System for Mechanical Assembly", *Proceedings of the IEEE International Conference on the Applications of Artificial Intelligence*, Orlando, FL, February 1987.

[Segre87b] A. M. Segre, "Explanation-Based Learning of Generalized Robot Assembly Plans", Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1987.