Evaluating and Debugging Analogically Acquired Models

Beth Adelson
Dept. of Computer Science
Tufts University
Medford, MA 02155 [1]

Abstract

We describe elements of a cognitive theory of analogical reasoning. The theory was developed using protocol data and has been implemented as a computer model. In order to constrain the theory, it has been developed within a problem-soving context, reflecting the purpose of analogical reasoning. This has allowed us to develop: A purpose-constrained mapping process which makes learning and debugging more tractable; An evaluation process that actively searches for bugs; And a debugging process that maintains functional aspects of base models, while adding target-appropriate causal explanations. The active, knowledge-based elements of our theory are characteristic of mechanisms needed to model complex problem-solving.

1. Introduction: Motivation and Goals

The research described here is part of an attempt to develop a cognitive theory of analogical reasoning. Because the purpose of analogical reasoning is to learn and solve problems, we have developed our theory within a problem-solving context. This approach provides powerful insights into the phenomenon and constraints on the theory.

Recent research suggests a class of theories which rests on the processes of retrieval, mapping, evaluation, debugging and generalization (Carbonell, 1983 & 1986; Falkenhainer, Forbus & Gentner, 1986; Holyoke & Thagard 1985 & in press; Burstein & Adelson 1987; Kolodner, 1985; Kedar-Cabelli, 1984). Our work extends existing cognitive theories of analogical reasoning by specifying the processes of mapping, evaluation and debugging as active; constrained by the problem-solving context; and dependent on knowledge about function, structure and mechanism [2]. In our theory, the mapping process can be focussed so that partial domain models, sufficient for the immediate problem-solving purpose, can be mapped. This incremental learning strategy renders both mapping and debugging more tractable (Adelson & Burstein, 1987). A detailed description of our theory of mapping can be found in Adelson (1989), here we focus on evaluation and debugging:

1. Active Evaluation: An intelligent problem-solver needs to be able to identify the bugs inherent in an analogically acquired domain model. Our system searches for bugs in newly mapped domain models. It does so by comparing the nature of the actions and objects in the newly mapped model to the nature of the actions and objects appropriate in the domain being mapped into. This allows the system to identify the aspects of the model that are inappropriate and therefore unlikely to hold in the domain being mapped into. Our system also has knowledge about the way in which analogical correspondences are meant to be understood across domains. These aspects of our system reflect powerful elements of human reasoning.

2. Active Debugging: Once the buggy portion of a model has been identified it must be replaced by a representation that is accurate in the new domain. Our system constructs and runs simulations in both source and target domains in order to identify mechanisms with *analogous functionality*. This allows the system to correct mapped models, maintaining the functionality provided by the analogical example while building a representation of a mechanism appropriate to the target domain. Here too our system's behavior characterizes the constrained way in which analogical examples are understood and used.

2. Protocol Data Illustrating the Issues

[2] The spirit of Carbonell's (1986) work on derivational analogy; Holyoke & Thagard's (in press) work on multiple constraint satisfaction; Kedar-Cabelli's (1984) work on purpose-guided reasoning, Burstein's (1983) work on causal reasoning and Falkenhainer. Forbus & Gentner's (1986) work on structure mapping is consonant with our view of analogical reasoning as an active, knowledge intensive process.

In developing our theory we have repeatedly drawn on the protocol data described below. These data yield insights into processes that need to be described in implementing a cognitive problem-solver.

The Protocol:

In collecting our data we video-taped a tutor teaching a student about stacks. The tutor's goal was to have the student be able to write Pascal procedures for pushing and popping items on to and off of stacks. At the beginning of the protocol session the student had just completed an introductory programming course in which he had learned about some basic programming constructs and about elementary data structures such as arrays and simple linked lists. (He had not learned about using a linked list as a stack.) The tutor had the intention of building upon the student's existing knowledge of Pascal through the use of analogy.

The relevant events of the protocol can be summarized as follows:

Learning about the behavior of a stack:

The tutor told the student that stacks are so named because their behavior is analogous to the behavior of the device that holds plates in a cafeteria. The student then proceeded to think of ways in which he might have previously encountered the use of stacks in programming; he suggested that stacks might be useful in implementing subroutine calls. He then stated that, in general, when a task had an unmet precondition it would be useful to delay execution of the task by pushing it onto a stack.

Learning about the mechanism underlying the behavior:

The tutor told the student that the mechanism of the computer science stack is in some sense analogous to the mechanism of the cafeteria stack. In order to achieve 'last in first out' (LIFO) behavior items are pushed *and* popped at the top of the cafeteria stack. The student drew a diagram of a cafeteria stack and noted that *push* causes the stack's spring to compress and *pop* causes it to expand.

Implementing push and pop in the target domain:

The student then wrote the code for *push* and *pop*. After writing *push*, however, the student asked if the capacity limitation which results when the spring is fully compressed is relevant in the new domain. The tutor told the student that the *physical* elements of the analogy (springs, movement of plates, etc.) do not apply. The student then asked if the concept of capacity limitation applies even if the spring doesn't. The tutor responded that although capacity limitation is an important concept, the student should disregard it for now.

3. Issues for Specifying a Theory of Analogical Problem-Solving

As diagrammed in Figure 1, the class of model to which our system belongs, contains a mapper, an evaluation and debugging mechanism, and a problem-solving component. The mapper takes as input a base domain model and a list of the correspondences between elements in the base model and already known target elements. The mapper produces tentative target domain models which are then debugged and evaluated. The debugged models are then used to in problem-solving. Additonally, in our system, the output of the debugger can be used to guide subsequent mappings.
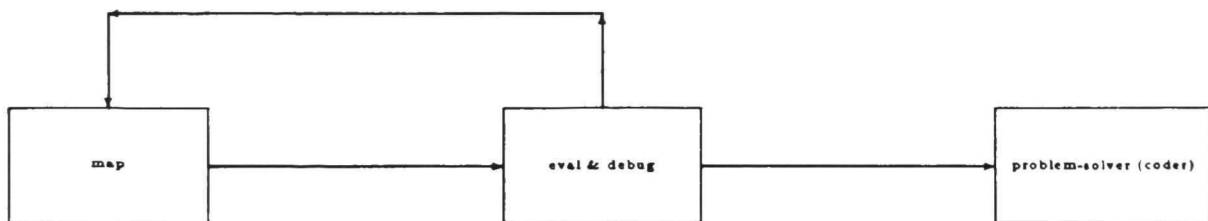

Figure 1: Components of the Analogical Reasoner

3.1. Purpose-Constrained Mapping

Purpose provides an essential constraint in problem-solving, but current implementations of cognitive theories do not make sufficient use of this constraint[3]. The argument for why purpose is necessary in constraining

---

[3] Thagard and Holyoke (in press), and Kedar-Cabelli, (1986) also stress the theoretical importance of purpose.

human problem-solving runs as follows. Understanding a complex domain requires understanding a number of distinct aspects of the domain and the relationships among those aspects (Burstein, 1986; Collins &Gentner, 1983; Adelson, 1984). Given the constraints of the cognitive system, it is not possible to learn all of these various aspects at one time. Rather, to make learning of a complex domain more tractable, students and instructors typically focus on individual, purpose-related aspects of the domain and, one at a time, map partial models from more familiar analog domains (Burstein & Adelson, 1987 & in press)[4]. In our computational description, the learning process starts with this selection and mapping of purpose-constrained aspects of the target domain. Our mapping mechanism focuses on partial models of the base domain whose type reflects the problem solver's purpose, and maps these models *separately*, type by type, over to the target domain[5].
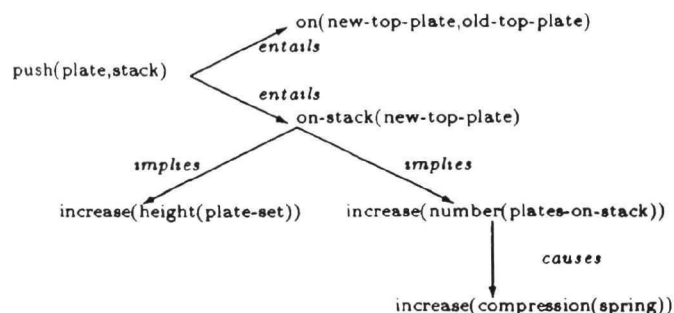
BASE: BEHAVIORAL
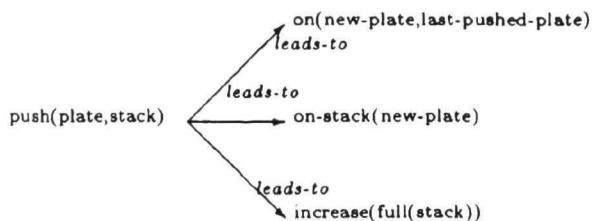
TARGET: BEHAVIORAL
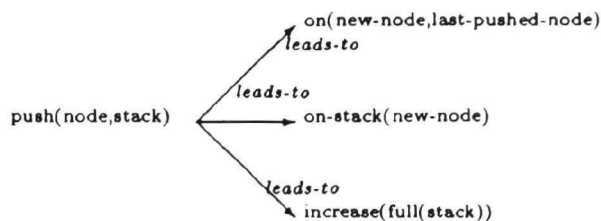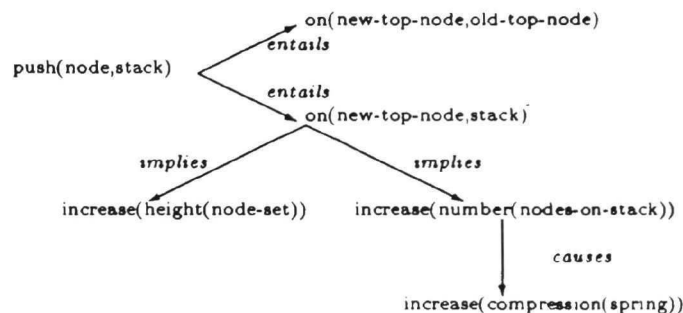
BASE: CAUSAL

TARGET: CAUSAL

Figure 2: Behavioral and Causal Models (top and bottom) in the Base and Target Domains (left and right)

The following illustrates a mapping produced by our system:

In this example our overall goal is to have the system model the problem-solving in our protocol. That is, we want the system to learn about computer science stacks and stack operations by analogy to cafeteria stacks and to produce the code for the operations *push* and *pop*. To accomplish this, the system begins by following the tutor's suggestion to focus initially on the behavioral model for *push*[6]. It selects that model from a base domain containing behavioral and causal models of both *push* and *pop*. The selected model is then mapped into the target domain. The behavioral and causal models for *push* in the base domain can be seen in the left half of Figure 2 (top and bottom respectively).

In the upper right of the figure we see the behavioral model of the target domain produced by our mapping mechanism. What is important to note here[7] is that the nature of the models allows them to be used in the problem-solving that is the system's ultimate goal. That is, the models in the figure describe the chains of events

---

[4] Our protocol data illustrates this process. The student acquired a sophisticated understanding of the behavior of a stack before learning about the mechanism supporting the behavior.

[5] See Burstein & Adelson (in press) for an explanation of the difference between models of varying types.

[6] The tutor also supplies a list of base and target domain correspondences stating for example, that *push* in the base corresponds to *push* in the target and plates correspond to nodes.

[7] Space limitations force us to omit several points about the models. The assertions in the model can be formalized to allow, for example, the deduction that the pushed node becomes the new top node. Additionally, the predicates do have an underlying

11

that occur when domain operations are performed (Schank & Riesbeck, 1981; Schank & Abelson, 1977). As a result, they can be run on the system's *simulation machine* (Adelson, 1989; Forbus, 1985; de Kleer & Brown, 1985). This allows them not only to be examined by a debugger (Sections 3.2.1 and 3.2.2) but also to be used to generate Pascal code after debugging.

The following section describes how our system debugs the newly mapped model of *push* so that problem-solving can be carried out successfully.

3.2. Debugging a Newly Mapped Model

3.2.1 Actively Seeking out Bugs

In our theory, debugging is characterized by an *active* search for bugs. The base domain model is known, by definition, to provide an imperfect model of the target domain. The base model may contain inappropriate elements that require deletion or transformation; or it may require additional knowledge specific to the target domain.

Our current example illustrates the case in which a newly mapped model contains a concept that is inappropriate in the target domain. Looking at the behavioral model in the target domain (Figure 2, upper right) we see that it pushing a node onto a stack which is implemented as a list of nodes, leads to the stack being more full. However, the system contains prior knowledge about the target domain which asserts that lists of nodes are used when a data structure without a pre-specified capacity limitation is desired[8]. Since linked lists have no specified capacity limitation there is an inconsistency between the newly mapped model and prior knowledge of the target domain. The system must have the ability to notice and resolve this inconsistency.

Identifying and Fixing Bugs in a Runnable Model:

Here we describe how the system's evaluation and debugging mechanism resolves the 'fullness' bug in the course of evaluating the behavioral model of *push*. The system's *evaluator* traverses a newly mapped model in an attempt to determine whether each element it encounters is appropriate in light of the domain the model has been mapped into. In order to allow the evaluator to carry out the evaluation the system has been given several kinds of knowledge:

K1. Any element that occurs in a model has a definition, a *template* consisting of a set of features (Burstein, 1983; Waltz, 1982; Winston, 1977 & 1982). Elements in the model are either objects (e.g. *stack*) or predicates which describe attributes of, or actions on the object (e.g. *fullness* or *push* respectively). For objects, one feature in the template specifies the class it belongs to. For predicates the class of both the predicate and its arguments are listed. For example, the predicate *full* is defined as a measurement of the capacity of some argument which must be a limited-capacity container.

K2. The system knows not only which objects and predicates are appropriate to each domain, but also which *classes* of objects and predicates. For example, the system knows that integer variables in particular, and data structures in general, are appropriate in the computer domain.

K3. The system has general knowledge about how analogical correspondences are meant to be taken. For example, the system contains knowledge that physical contiguity in the base can be appropriately thought of as corresponding to virtual contiguity in the target.

The system uses the knowledge described above in applying rules which allow it to evaluate each element in the newly mapped model. The rules are:

R1 Infer that an element currently in the target domain is appropriate in the new model.

R2. If the element is not currently in the target domain but it is of a *class* currently in the target domain. infer that a modified version of the element is appropriate in the new model and use existing domain information

---

semantics. For example, the value of full results from dividing the current number of plates by the maximum number of plates. Along these lines, the models in the figure have substantially less detail than the actual models used by the system.

[8]For this example we have supplied the system with the same knowledge of the target domain that our novice programmer had. We have given it models for performing typical operations on variables, arrays and linked lists. It also has world knowledge about boxes and containers.

12

about the class to modify the element[9].

R3. If the element belongs to a class that has a corresponding class in the target (point K3), infer that a modified version of the element belongs and use existing domain information about the coresponding class to modify the model.

R4. A predicate can only be applied to an argument of an appropriate type. That is fullness cannot be predicated of a container without capacity limitations (point K1).

We see each of the above rules (and the knowledge they embody) being applied as we follow the evaluator tracing through the model for *push*. Starting at the root of the tree, the evaluator encounters the element *push*. Because the tutor had specified that *push* in the base corresponded to an asserted, but as yet undefined, version of *push* in the target the system infers that *push* is appropriate to the model and turns to the predicates that follow from it.

The template for the predicate *on* states that it is a "physical contiguity relation". The system knows that physical contiguity in the base corresponds to virtual contiguity in the target (Rule R3). It therefore makes this change to the predicate's template and then infers that the predicate holds. This is an example of the system's ability to interpret analogical correspondences in an appropriate, non-literal manner.

The evaluator comes to the next predicate that *push* leads to. Although *on-stack* does not yet exist in the target, its definition states that it is a "membership relation". The system finds that other predicates in the target are membership relations (e.g. *in-set*) and therefore hypothesizes that the predicate holds (Rule R2).

The evaluator turns to the predicate *full*, it finds that *full* is potentially appropriate in that it already exists in the target as knowledge that arrays can be full (Rule R1). However, the evaluator finds that fullness can only be predicated of containers having capacity limitations (Rule R4). It knows that the stack is being implemented as a list of nodes and that lists do not have capacity limitations. The system suggests that the concept fullness should be removed from the model. It then removes fullness. At this point, if the system is told that the problem arose because no capacity limited containers were being used in this example it will also remove all other predicates whose definitions involve capacity limitations.

But more mileage can be gotten out of this evaluation. The system has just mapped and debugged the behavioral model. It can now to go back and map the causal model using information gained in debugging the behavioral one. When this mapping begins the system will take note of any elements that have been deleted from the behavioral model (in this example, *full*); pieces of the causal model that only support an element already deleted from the behavioral will not be mapped. As a result of our strategy of incrementally mapping partial models and using earlier mappings to guide subsequent ones, the debugging of potentially complex causal models can be made considerably simpler.

3.2.2 Transforming a Mapped Model: Reasoning About Simulations

In the example just described we considered the case where an element needs to be deleted. Our system also handles the case in which a model needs to be corrected by being *transformed* through finding additional correspondences between elements of the base and target domains.

The case in which a model needs this type of transformation is illustrated by the example in which the goal is to implement a stack using an array rather than a linked list. The representation of the base domain is the same as it was for our earlier example. Prior knowledge of the target domain still consists of information about variables and arrays but not about linked lists. The system again has runnable models for typical operations such as initialization and search.

The behavioral model of push is again mapped into the target domain. This time no changes are made in the behavioral model; the fullness of the stack is found to be consistent with the system's knowledge of the capacity limitation of an array. After mapping the behavioral model, the system maps the causal model of the stack into the target domain and then begins to evaluate and debug it. During this process the system questions the tutor on the appropriateness of the spring in the causal model (see Figure 2, bottom right). The tutor tells the system that the domain-appropriate *functional analog* of the spring needs to be found. In finding the functional analog

---

[9]Space constraints preclude a description of some of the methods that have been developed to modify 'almost right' elements.

13

of the spring the system will draw on several types of *relational knowledge* which comprise the system's causal ontology[10]:

RK1. The system contains functional to structural mappings; knowledge relating state changes and the mechanisms causing them (Adelson, 1984; de Kleer & Brown, 1985; Forbus, 1985; Kuipers, 1985). For example, it knows that 'changes in fullness are supported by changes in the mechanism comprised of the spring, the set of plates, etc'[11].

RK2. The system has knowledge relating actions and the state changes they produce. It knows, therefore that 'pushing leads to changes in fullness'

RK3. The system also has knowledge relating actions and the mechanisms involved. It knows that 'pushing involves a change in the spring'

In order to find the piece of target domain mechanism with the same function as the spring, the system will find what sort of state change in the base is associated with a particular change in the spring. It will then turn to the target look at the parallel state change and determine what piece of mechanism is effected in the way that the spring was. To do this the system first needs to focus on the base and find what state changes the spring is involved in. It examines its knowledge of functional to structural mappings (RK1) and finds that the spring is involved in changes in fullness. Now, in order to find out the nature of the relationship between changes in fullness and changes in the spring, the system looks for a simulation in which both fullness (RK2) and the spring (RK3) will change. It finds that simulating *push* will produce the needed information. The simulation is run, providing the system with values for the fullness of the stack and the compression of the spring before and after the simulation is run. The system then compares the direction of change in both fullness and spring compression and finds that there is a positive relationship between the two [12].

The system now needs to find what piece of mechanism in the *target* domain changes for the same reason and in the same way as the spring (i.e. increases with fullness). The system begins by looking for an operation in which fullness increases. It will then run this operation and look for pieces of mechanism that register increases in fullness.

Target domain knowledge about the relation between actions and state changes (RK2) asserts that initializing an array causes fullness to increase; the system simulates the process and finds that in the target, it is the array-index that increases with fullness. As a result of this process, in which corresponding simulations are sought, run and evaluated for the purpose of finding functionally analogous mechanisms the system correctly hypothesizes that the array index is the analog of the spring[13]

Space limitations prevent further description of our system, but the debugging process does not end here. Now that the models mapped from the base domain have had changes made to them, they must be checked to see that they are still sufficient. This is done through a series of simulations designed to test that the models still exhibit aspects of LIFO behavior that the system knows are important. For example, pushing and then popping a set of elements must result in reversing their ordering. Additionally, the system must eventually produce both box and arrow and pascal versions of push and pop (Adelson et al, 1988).

4. Summary & Conclusions

we have presented a discussion of three of our system's mechanisms: one for mapping, one for evaluating mapped models and one for debugging inconsistencies. We have implemented a purpose-constrained mapper that reflects the way students limit their focus of attention. The strategy results in incremental learning which makes both mapping and debugging more tractable. We have also implemented an evaluation mechanism that identifies inconsistencies as elements of newly mapped models are checked to see if they are the sort of elements that are known to exist in the target domain. In doing so the evaluation mechanism uses knowledge about the nature

---

[10] These relations are learned in that the system notices and stores these types of relations whenever it acquires a new model.

[11] The system's knowledge does not contain any explicit statement concerning *how* changes in fullness are related to changes in the spring. This is what needs to be determined.

[12] Currently the system can recognize positive and negative correlations, as well as the lack of relationship between two state variables. It is possible to expand this part of the system to include the recognition of more complex, but regular relationships.

[13] When more than one piece of mechanism is found, the system has the ability to use functional information decide on the better analog.

14

of the base and target domains and they way in which relations apply across analogous domains. Finally, we have presented a debugging mechanism that maintains functional aspects of base models while adding target-appropriate causal explanations. The development of the mechanisms has been possible because we have worked within a problem-solving context, reflecting the purpose of analogical reasoning.

5 References

Adelson, Beth. Cognitive modeling: Uncovering how designers design. *The Journal of Engineering Design.*. Vol 1,1. 1989.

Adelson, Beth. When novices surpass experts: How the difficulty of a task may increase with expertise *Journal of Experimental Psychology: Learning, Memory and Cognition*, July 1984.

Adelson, B., Gentner, D., Thagard, P., Holyoak, K., Burstein, M., and Hammond, K. The Role of Analogy in a Theory of Problem-Solving *Proceedings of the Eleventh Annual Meeting of the Cognitive Science Society*, 1988.

Burstein, Mark H. Causal Analogical Reasoning. *Machine Learning: Volume I*. Michalski, R.S., Carbonell, J.G. and Mitchell, T M. (Ed.), Los Altos, CA: Morgan Kaufman Publishers, Inc., 1983.

Burstein, Mark H. Concept Formation by Incremental Analogical Reasoning and Debugging. *Machine Learning: Volume II*. Michalski, R.S., Carbonell, J.G. and Mitchell, T M. (Ed.) Morgan Kaufmann Publishers, Inc., Los Altos, CA 1986.

Burstein, M. and Adelson, B. Mapping and Integrating Partial Mental Models. *Proceedings of the Tenth Annual Meeting of the Cognitive Science Society*, 1987.

Burstein, M. and Adelson, B. Analogical Reasoning for Learning. in *Applications of Artificial Intelligence to Educational Testing*. R. Freedle (Ed.) In press. Erlbaum: Hillsdale, NJ.

Carbonell, Jaime G. Transformational Analogy Problem Solving and Expertise Acquisition. *Machine Learning: Volume I*. Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. (Ed.), Los Altos, CA: Morgan Kaufman Publishers, Inc., 1983.

Carbonell, Jaime G. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. *Machine Learning: Volume II*. Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. (Ed.), Los Altos, CA: Morgan Kaufman Publishers, Inc., 1986

de Kleer, J. and Brown, J. S. A Qualitative Physics based on Confluences In *Qualitative Reasoning about Physical Systems* D Bobrow editor, MIT Press 1985.

Falkenhainer, B., Forbus, K. and Gentner, D. The Structure-Mapping Engine In *Proceedings of AAAI-86*. Los Altos, CA: Morgan Kaufman, 1986.

Forbus, Ken. Qualitative Process Theory In *Qualitative Reasoning about Physical Systems* D. Bobrow editor, MIT Press 1985

Gentner, Dedre. Structure-Mapping: A theoretical framework for analogy. *Cognitive Science*, 1983, 7(2), 155-70.

Holyoke, K. and Thagard, P. Analogical Mapping by Constraint Satisfaction. Cognitive Psychology. In Press.

Kolodner, J. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*. Boulder, CO: Cognitive Science Society, 1985.

Kuipers, B. Commonsense Reasoning About Causality. In *Qualitative Reasoning about Physical Systems* D. Bobrow editor, MIT Press 1985.

Kedar-Cabelli, S. Analogy with Purpose in Legal Reasoning from Precedents. Technical Report 17. Laboratory for Computer Science, Rutgers. 1984.

Schank, R., and Abelson, B. Scripts, Plans, Goals and Understanding. Hillsdale, NJ: Erlbaum, 1977.

Schank, R. and Riesbeck, C. Inside Computer Understanding. Erlbaum: Hillsdale, NJ. 1981.

Waltz, D. Event shape diagrams. In Proceedings of the National Conference on AI. 1982.

Winston, P. Learing new principles from precedents and exercises. *AI*. 1982.