UC Merced
Proceedings of the Annual Meeting of the Cognitive Science Society

Title
A Framework for Opportunistic Abductive Strategies

Permalink
https://escholarship.org/uc/item/3bz8c4v5

Journal
Proceedings of the Annual Meeting of the Cognitive Science Society, 13(0)

Authors
Johnson, Todd R.
Smith, Jack W.

Publication Date
1991

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A Framework for Opportunistic Abductive Strategies*

Todd R. Johnson

Laboratory for Artificial Intelligence Research
Dept. of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210
Email: tj@cis.ohio-state.edu

Jack W. Smith

Division of Medical Informatics
The Ohio State University
571 Health Sciences Library
376 W. 10th Ave.
Columbus, Ohio 43210
Email: smith.30@magnus.ohio-state.edu

Abstract

Any single algorithm for abduction requires specific kinds of knowledge and ignores other kinds of knowledge. A knowledge-based system that uses a single abductive method, is restricted to using the knowledge required by that method. This makes the system brittle, because the single fixed method can only respond appropriately in a limited range of situations and can only make use of a subset of the potentially relevant knowledge. In this paper, we describe a framework from which abductive strategies can be opportunistically constructed to reflect the problem being solved and the knowledge available to solve the problem. We also describe ABD-Soar, a Soar-based implementation of the framework, and demonstrate its behavior.

Introduction

Abduction, finding a best explanation for a set of data, is an important part of many knowledge-based (KB) systems, particularly those concerned with diagnosis. In recent years, several different algorithms for doing abduction have been devised. Any single algorithm for abduction, however, requires specific kinds of knowledge and ignores other kinds of knowledge. Hence, a KB system that uses a single abductive method, is restricted to using the knowledge required by that method. This makes the system brittle, because the single fixed method can only respond appropriately in a limited range of situations and can only make use of a subset of the potentially relevant knowledge. To remedy this we have endeavored to develop a framework from which abductive strategies can be opportunistically constructed at run-time to reflect the problem being solved and the knowledge available to solve the problem. In this paper, we present this framework and describe ABD-Soar, an implementation of the framework. We show how ABD-Soar can be made to behave like the abductive strategy used in Red (Josephson, et al., 1987), a system for red cell antibody identification. Finally, we discuss the differences between ABD-Soar and 2 tools for abduction: Peirce (Punch III, et al., 1990), and Mole(Eshelman, 1988).

This work contributes both to our understanding of KB systems and abduction. First, it illustrates how the problem-solving capabilities of KB systems can be increased by using mechanisms that permit the use of all relevant knowledge. ABD-Soar requires little domain knowledge to begin solving a problem, but can easily make use of additional knowledge to solve the problem better or faster. Second, the framework can be used to provide a flexible abductive problem-solving capability for KB systems. Third, ABD-Soar gives Soar an abductive capability such that many systems written in Soar can begin to solve abductive problems. Fourth, the framework provides a very simple and general mechanism for abduction that is capable of generating the behavior of various fixed methods. Fifth, ABD-Soar can be used to experiment with different abductive strategies, including variations of existing strategies and combinations of different kinds of strategies. Finally, the framework provides guidelines for building abductive systems because it provides a theory of abduction and specifies the kinds of knowledge needed to do abduction.

* This research is supported by National Heart Lung and Blood Institute grant HL-38776, and National Library of Medicine grant LM-04298.

The Abductive Task

Most Artificial Intelligence research on abduction is concerned with strategies for a type of abduction called *hypothesis assembly*. In hypothesis assembly, a complete explanation of the data must be formed by composing a number of smaller explanations. For example, in medical diagnosis the data to be explained are the signs and symptoms of the patient. If the patient has multiple diseases, the best explanation for the data will consist of all of the diseases present. Each disease might only explain a subset of the data, but taken together they must explain all of the data.

Most models of abduction can be described using four criteria to define the best explanation: *coverage* (the number of explanata[1] explained or covered by the explanation), *belief* (some measure of plausibility), *parsimony*, and *consistency* (Josephson and Goel, 1988). The models vary in their interpretation of these criteria. For instance, parsimony can be defined in terms of *minimal cardinality*, i.e., the fewest number of component hypotheses, or in terms of *irredundant covers*, i.e., an explanation in which no proper subset of the component hypotheses can explain the data. Belief is also subject to different interpretations. Some models view the most plausible explanation as that with the highest overall plausibility. Other models view the most plausible explanation to be that in which each component hypothesis is the most plausible explanation for at least one datum (this has been called *best-for-some* (Bylander, et al., 1988)). In general, there is probably no single model of abduction appropriate for all abductive tasks. Ultimately, the goals of the specific abductive task and characteristics of the task domain define what counts as a best explanation.

An analysis of the computational complexity of abduction has shown that the problem is, in general, computationally intractable (Allemang, et al., 1987). There are, however, some conditions under which the problem is computationally tractable. Bylander, et. al. writes:

> Our primary conclusion is that very restrictive conditions must be satisfied for abduction to be tractable: determining the plausibility and explanatory cover-

[1] An *explanatum* is a datum to be explained. *Explanata* are all of the data to be explained.

age of hypotheses must be tractable, there cannot be substantial incompatibility and cancellation interactions between hypotheses, and plausibility comparison between composite hypotheses must be logically weak. Specific domains may escape some of these requirements if composite hypotheses are guaranteed to be small or if strong domain knowledge can rule out most individual hypotheses. (Bylander et al., 1988)

Thus, it is very unlikely that any single strategy for abduction is appropriate to all tasks. The efficient solution of most real-world abduction problems requires the careful application of domain knowledge to transform the problem into either one that is tractable or one that is small enough so that tractability concerns are unimportant. Hence, it is very important to build systems that can make use of all of the available knowledge when solving abductive problems.

Specifying Problem-Solving Methods

Before we describe the framework for abduction, we must first explain how we are going to specify it. Here we address this in the context of the general problem of specifying problem-solving methods.

One view of a problem-solving method is that it consists of a set of operators (operations on data) with preconditions and knowledge that indicates the order in which to apply those operators. Since we desire an opportunistic system, we need to be able to specify a set of operators without necessarily specifying a complete ordering of those operators. An opportunistic system works by enumerating possible operators to apply to the immediate situation and then selecting one of those operators based on the current goal and situation. Furthermore, a flexible system must be capable of generating or using additional control knowledge. Thus, if a system must decide between several operators, it must be possible for the system to engage in complex problem-solving to determine which operator is best.

To achieve these results we have been using the problem-space computational model (PSCM) (Newell, 1990) to specify methods. In the PSCM, all problem-solving is viewed as search for a goal state in a problem-space. Knowledge about when operators are applicable to a state can be specified independent of knowledge about which operator to select. Operator selection knowledge, called *search control knowledge*, is expressed in terms of preferences for or against applicable operators. If at any time during the problem-solving the search control knowledge is insufficient to indicate which operator to select, a subgoal is set up to generate additional knowledge so that a single operator can be selected. This subgoal is achieved by searching another problem-space. Operators can either be implemented by directly available knowledge or by using an operator-specific problem-space. Implementation in a problem-space is similar to using a subfunction to implement an operator in lisp.

The Framework

The general abductive framework can be described by a single problem-space its goal, the knowledge content of its states, the initial state, operators, and search control knowledge. Following the description of the problem-space, we describe the minimal knowledge required to use the framework.

State Description: The state contains knowledge of explanata, explanations and an indication of the explanata they explain, knowledge about inconsistent or redundant objects, information about whether an explanation is the only possible way to explain an explanatum, and knowledge about whether the implications of a newly added object have been processed. In addition, any other information necessary for solving the problem can be kept in the state.

Initial State Schema: The initial state need contain only the explanata. Additional information can be provided.

Desired State Schema: The desired state must meet 6 conditions: 1) The explanation must be *complete*, i.e., all the explanata must be explained. 2) The hypotheses in the explanation must be at the desired level of detail for the problem being solved. For example, in diagnosis a disease hypothesis must be at a level of detail such that a therapy can be recommended. 3) No part of the explanation can be redundant. 4) No part of the explanation can be inconsistent. 5) All parts of the explanation must be certain. 6) All parts of the explanation must be processed. That is, the implications of adding the object and its effect on the rest of the explanation must have been considered.

There are 7 operators in the abductive space. The last 3 are used to determine the logical implications of adding an object to the explanation. 1) **Cover *explanatum*** is proposed for each explanatum that is not yet explained by a hypothesis (the explanation is not yet complete). Its goal is to add to the composite explanation one or more hypotheses that explain the explanatum. 2) **Resolve-redundancy** is proposed whenever there are redundant objects in the explanation. Its goal is to make the explanation irredundant. 3) **Resolve-inconsistency** is proposed whenever there are inconsistent objects in the explanation. Its goal is to make the explanation consistent. 4) **Determine-certainty *object*** is proposed for each uncertain object in the explanation. The operator is successfully applied when the object is deemed to be certain, or else the object is deemed to be not present in the explanation. 5) **Determine-accounts-for *hypothesis*** is proposed for each new hypothesis. Its goal is to determine what explanata the hypothesis can account for. 6) **Mark-redundancies *object*** is proposed for each new object. Its goal is to indicate which objects in the explanation are redundant with the newly added object. Hypotheses that offer to explain an identical explanatum (or explanata) are considered redundant unless other knowledge indicates that they are not. 7) **Mark-inconsistencies *object*** is proposed for each new object. Its goal is to indicate which objects are logically inconsistent with (or contradict) the newly added object.

Search control knowledge is specified as 1) *Determine-accounts-for* is better than all other operators. 2) *Resolve-redundancy, resolve-inconsistency,* and *determine-certainty* are indifferent to each other. [Note that an order can still be imposed on these operators by preferring one over another for a particular domain.] 3) *Mark-redundancies* and *mark-inconsistencies* are equal to one another and better than all other operators except for *determine-accounts-for*.

This is the minimum search control needed to ensure correct operation of the abductive mechanism. However, this search control only specifies a partial ordering of the operators. Any control decisions that can be based on domain-dependent knowledge have been left unspecified. This allows the designer of the system to add appropriate search control for the task being done. For example, there is no knowledge about what to do when multiple cover operators tie since this decision can be based on domain-dependent knowledge.

To use the framework, a minimum of 6 kinds of knowledge is required: (1) knowledge mapping each possible explanatum to potential explanations (to implement *cover*); (2) knowledge mapping an explanation to the explanata it can explain (to implement *determine-accounts-for*); (3) knowl-

edge to determine the certainty of the explanations (to implement *determine-certainty*); (4) knowledge about the consistency of the model (to implement *mark-inconsistencies*); (5) knowledge about redundant objects (to implement *mark-redundancies*); and (6) additional search control knowledge to sequence the operators.

As a result of the operators and search control, the basic method is to pick an explanatum to cover, add one or more explanations for that explanatum, determine what each new explanation explains, and then pick another (unexplained) explanatum to explain. This continues until all explanata are explained. If at any time the model becomes inconsistent, redundant, or uncertain, an operator is proposed to resolve the problem. At that time, a decision must be made about whether to fix the problem or continue covering explanata.

Implementing the Framework: ABD-Soar

ABD-Soar is a Soar-based implementation of the abductive framework. Soar is used because it directly supports the PSCM; however, the framework does not absolutely require Soar—other architectures that support PSCM-like functionality can also used. ABD-Soar supplies all of knowledge specified in the framework: knowledge to propose the abductive operators and detect their successful application, and the minimal search control specified in the framework. This knowledge is encoded as a set of Soar productions that can apply to any problem-space. This means that the complete body of abductive knowledge can be brought to bear during any problem-solving activity. ABD-Soar also provides default implementation knowledge for *cover*, *resolve-redundancy*, and *resolve-inconsistency* and a default method for generating additional search control knowledge.

The default knowledge for operator implementation is encoded in three problem-spaces with names identical to the operators they implement: *cover*, *resolve-redundancy*, and *resolve-inconsistency*. *Cover* generates possible explanations and then applies knowledge to select one of the candidates. The candidates are generated in response to an *explain* operator that must be implemented using domain specific knowledge. *Resolve-redundancy* and *resolve-inconsistency* remove each redundant/inconsistent object until the explanation is irredundant/consistent. If multiple irredundant/consistent explanations are possible, then all are found and the best one is used. This makes use of lookahead and the evaluation function described below. Any explanation that is an *absolute essential* (i.e., the only possible explanation for an explanatum) will not be removed. Also, it is possible to resolve a redundancy by explicitly indicating that particular redundant objects are not a problem.

The default method for generating additional search control knowledge is to use lookahead and an evaluation function to determine which operator to take when multiple operators are applicable. This is implemented in two problem-spaces: *find-best* and *evaluate-state*. *Find-best* evaluates each operator and selects the operator with the best evaluation. To evaluate an operator *find-best* applies the operator to a copy of the original state and then continues to do problem-solving from that state until a state is found that can be evaluated. The default evaluation function for *evaluate-state* is a summation of the number of explanata left to explain, the number of explanations, the number of inconsistent objects, the number of explanata explained by inconsistent explanations, the number of redundant objects, and the number of uncertain objects. The model with the lowest evaluation is chosen as the best alternative.

ABD-Soar uses irredundant covers as the parsimony criterion; however, when the default evaluation function is used the system also uses minimal cardinality to choose between competing composite explanations. The belief criterion (by default) is best-for-some since the method attempts to pick the best explanation for a finding being covered. In the absence of plausibility ratings, the evaluation function is used to rate competing explanations for a finding and the one with the best evaluation is selected.

Red Cells			
1	2	3	4
(r1)1+	(r2)3+	(r4)1+	(r5)1+
0	(r3)1+	0	0

Anti-K explains r2 and r3, ie, the 3+ and 1+ on red cell 2
Anti-Fy^a explains r1, r4, and r5, ie, the 1+ on red cells 1, 3, and 4
Anti-C explains r1 and r4, ie, the 1+ on red cells 1 and 3
Anti-N explains r4, ie, the 1+ on red cell 3
0 reactions do not need to be explained

Figure 1: Red cell antibody identification case

To use ABD-Soar the designer of a system must provide knowledge to implement *explain* and *determine-accounts-for*. When necessary, knowledge must also be provided for *mark-inconsistencies*, *mark-redundancies*, and *determine-certainty*. Given just this knowledge and the built-in defaults, abductive problems can be solved. An example of the behavior produced under these conditions is given below in the section on demonstrating ABD-Soar.

Optionally, the designer can choose to add additional search control knowledge and/or knowledge to override any of the default knowledge. Adding knowledge beyond the minimum requirements can greatly increase the efficiency of the abductive system. ABD-Soar is completely open with respect to the addition of new knowledge. Additional operators can be added to any space. New ways of implementing existing operators can be added. Search control knowledge can be added so that lookahead can be avoided. Furthermore, the additions can be made general so that they work for all tasks, or specific so that they only work for a single task or problem. Every addition of knowledge will alter problem-solving behavior. In this framework the available knowledge shapes the strategy, unlike the traditional approach where strategies must be designed to use pre-specified kinds of knowledge.

Demonstration of ABD-Soar

The problem-solving behavior of ABD-Soar changes according to the task and the knowledge available to solve the task. Here we describe the default behavior of the system and show what knowledge must be added so the system will behave similar to Red's strategy for abduction. The first example illustrates the system's default behavior. The second example illustrates behavior similar to that of Red.

Both of the examples use the red cell antibody identification case shown in Figure 1. The details of the domain are unimportant—only the knowledge of what each antibody explains is necessary to understand the examples. The correct answer for this case is anti-K and anti-Fy^a.

Example 1: Default Behavior

To solve the case in Figure 1 using default behavior requires a domain space with an initial state containing the reactions (the explanata) and knowledge to implement *explain* for each reaction and *determine-accounts-for* for each antibody.

Explain is implemented using a single production for each reaction and explanation pair. These are of the form: *if trying to explain 1+ on cell 1 then consider Anti-Fy^a*. *Determine-accounts-for* requires two productions for each antibody: one to enumerate the reactions explained by the antibody and one to detect that the operator has been applied. Overall, the case consists of 31 productions.

A fragment of the system's behavior is shown in Figure 2. The numbers indicate the problem-solving sequence and are used in the following description to indicate what part of the figure is being discussed. Five cover operators (one for each reaction) are applicable to the initial state. This leads to an *operator tie* impasse (1) and the selection of *find-best* to break the tie (2). Find-best evaluates each operator by applying it to a copy of the initial state of the domain space. The figure shows a fragment of the lookahead process for *cover r2* (3). The domain space is selected (4) and *cover r2* is applied to the state (5). The cover operator is implemented in the *cover* space (6) by applying *explain* (7) which generates a single explanation and then *add* (8) to add that explanation to the model. Once anti-K is added, *cover r2* has been successfully implemented (9). Lookahead problem solving then continues in the domain space by processing the newly added explanation (10) using four operators: *determine-accounts-for*, *mark-redundancies*, *mark-inconsistencies*, and *new-to-processed*. Their location is indicated with an ellipse in the figure. Next, three cover operators can apply to the current state so an *operator tie* impasse arises (11). This impasse is resolved using lookahead (not shown) resulting in the selection of *cover r4*. This reaction is explained using anti-Fy^a which, when processed (12), results in a model that satisfies the abductive criteria. At this point the state is evaluated (13) and the result of the evaluation is returned to the *find-best* space (14). The system then repeats this process with the remaining *cover* operators (15-18). In this example, all of the *cover* operators evaluate to 2. This means that no matter which finding is picked to *cover* first, the resulting explanation will be equally good. Based on these evaluations *find-best* generates *indifferent* preferences for each of the *cover* operators (19). This allows the system to choose a *cover* operator at random. In this run, the system decides to cover the 1+ on red cell 1 (20). This reaction can be explained by anti-Fy^a or anti-C, so the system uses lookahead to rate each explanation (not shown). Anti-Fy^a results in a better evaluation since it explains more than anti-C, so anti-Fy^a is added to the model and processed (21). Next the system randomly chooses to cover the 3+ on red cell 2 (22). Anti-K is the only antibody that will explain this reaction, so it is added to the model and processed (23). This results in a best explanation that satisfies all of the abductive criteria so the system halts. A total of 1038 decision cycles were required to solve the problem. A decision cycle corresponds to the selection of a goal, problem-space, state, or operator.

This example illustrates how lookahead can be used to generate knowledge about what to do when the designer of the system does not or cannot supply that knowledge. However, lookahead is quite expensive so it is desirable to add additional knowledge whenever possible. The next example illustrates how the addition of some simple search control knowledge can greatly decrease the number of decision cycles needed to solve the problem.

Example 2: Red-like Abduction

Red uses two heuristics to help prune the search space: (1) The system prefers to cover stronger reactions before weaker reactions. If reactions are equal, then one can be selected at random to cover; and (2) Whenever multiple antibodies can explain the same reaction, the antibodies are ordered according to plausibility and the one with the highest plausibility is used. If multiple antibodies have the same plausibility, then one is selected at random. This knowledge can be added to ABD-Soar by rating antibodies with a plausibility and by adding three search control rules: ID1) If $r_1 > r_2$ then *cover* r_1 is better than *cover* r_2. ID2) If r_1 is equal to r_2 then *cover* r_1 is indifferent to *cover* r_2. ID3) If a_1 has a higher plausibility than a_2 for explaining a reaction then a_1 is a better explanation than a_2. Since it has not been specified that equally plausible antibodies are indifferent, the system will do lookahead to differentiate between them.

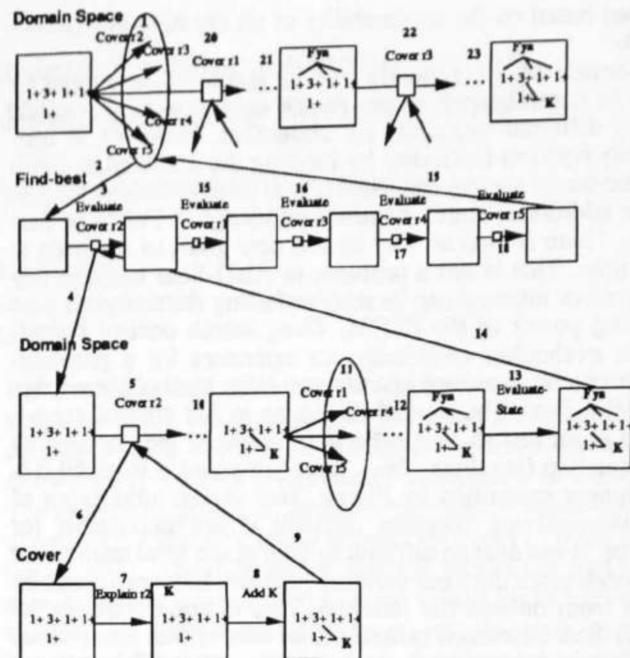

Figure 2: Default behavior of ABD-Soar

As a result of this knowledge, the system solves the case in 23 decision cycles (versus 1038 for example 1). First the system decides, because of ID1, to cover the 3+ on red cell 2. It does this using anti-K, the only antibody that explains the 3+. Next, because of ID2 the system randomly selects the 1+ reaction on red cell 1 to explain. This can be explained using either anti-Fy^a or anti-C, but anti-Fy^a is more plausible so, because of ID3, anti-Fy^a is selected.

This example illustrates how small changes to search control knowledge can radically alter the behavior of the system. The system in this example exhibits behavior very much like Red. Furthermore, whenever the search control is inappropriate, the system can fall back on the default knowledge to make progress.

Comparison to Other Tools

Peirce

Peirce is a tool built to do hypothesis assembly (Punch III, et al., 1990). It provides a control mechanism, called *sponsors and selectors* (Brown and Chandrasekaran, 1989), to encode search control knowledge for sequencing and achieving its abductive goals in a somewhat flexible manner. Flexible subgoal sequencing is accomplished by using knowledge about the applicability of subgoals to the current situation (*sponsors*) and knowledge about how to choose a subgoal to

pursue based on the applicability of all the subgoals (*selectors*).

Because of the generality of its goal/subgoal hierarchy and its control mechanism, Peirce can be used to encode many different strategies for abduction. However, it ultimately restricts flexibility by limiting the knowledge that a Peirce-based system can use. First, there is no way to generate additional search control knowledge in Peirce at run-time. There is also no way to add new goals or methods at run-time. This is not a problem in ABD-Soar because any subgoal or impasse can be resolved using the complete processing power of the PSCM. Thus, search control knowledge, evaluation knowledge, or operators for a problem-space can be generated just like any other kind of knowledge in ABD-Soar. The second limitation is that control knowledge is limited in Peirce because methods are encoded as opaque lisp functions. Thus, once a method is invoked it is no longer controlled by Peirce. This makes monitoring of problem-solving progress difficult if not impossible for Peirce. It can also be difficult to look at the local state of the methods since the local variables of a lisp function cannot be seen from outside the function. This is not a problem for ABD-Soar because it is meant to be used within Soar, where all "methods" are implemented within the architecture as problem-spaces whose states are accessible by knowledge associated with any problem-space. Finally, the goal/subgoal structure in ABD-Soar is much finer grained than the one used in Peirce. This means that the abductive strategy can be controlled at a finer level of detail in ABD-Soar.

Mole and Cover-and-Differentiate

Cover-and-differentiate (McDermott, 1988) is a method for a form of abduction. It is implemented in Mole (Eshelman, 1988), a tool for building cover-and-differentiate systems. McDermott defines its method as 1) Determine the events that potentially explain the symptoms. 2) If there is more than one candidate explanation for any event, then identify information that will differentiate the candidates by ruling out an explanatory connection, ruling out an explanatory event, confirming an explanatory event, or preferring one explanatory connection over another. 3) Get this information and apply it (in any order). 4) If step 3 uncovers new symptoms, go to step 1(McDermott, 1988).

This method and Mole, in particular, have several limitations from a flexibility perspective. First, the method is extremely rigid—it specifies a specific fixed sequence of actions for solving the problem. Hence the search control knowledge is pre-specified and cannot be altered or extended. Second, the "grain-size" of the actions are quite large. For example, all possible explanations for every finding must be enumerated in a single step. This eliminates all methods that decompose the problem into a number of smaller problems. This, in turn, means that knowledge about what to cover first (something that can increase the efficiency of problem-solving) cannot be used by the method. Third, all possible candidate explanations must be statically pre-enumerated—Mole does not allow the candidates to be generated or constructed at run-time. Finally, the method is most applicable when there is only a single fault—the method is not designed for hypothesis assembly.

Limitations

ABD-Soar has two limitations. First, to use ABD-Soar, a system builder must be well acquainted with Soar. Second, the current version of ABD-Soar does not use Soar's built-in learning mechanism. This is because the low level representation of annotated models we used causes Soar to learn overgeneral productions. This can be solved by using a different representation for annotated models. Such a version of ABD-Soar is being planned.

Conclusion

ABD-Soar implements an extremely general framework for building abductive systems. It can use a wide range of knowledge and alter its behavior based on that knowledge. Instead of programming a method for abduction, a system builder can give the system the knowledge available to solve the problem and the system will behave appropriately. In the absence of specific knowledge the system can fall back to default knowledge to make progress, however slowly. Along with its flexibility and generality, ABD-Soar provides guidance for building systems and acquiring knowledge because it provides a theory that specifies specific kinds of domain and search control knowledge.

Acknowledgments

We thank B. Chandrasekaran, John Josephson, and Kathy Johnson and the AIM research group for their assistance and comments on this paper and the work it reports. We also thank the members of the Soar community for their intellectual and technical support.

References

Allemang, D., Tanner, M., Bylander, T., &Josephson, J. R. 1987. On the Computational Complexity of Hypothesis Assembly. In Procs. of the Tenth International Joint Conference on Artificial Intelligence, :1112-1117. Milan, Italy.

Brown, D. C., &Chandrasekaran, B. 1989. *Design Problem Solving: Knowledge Structures and Control Strategies*. San Mateo, CA: Morgan Kaufmann Publishers.

Bylander, T., Allemang, D., Tanner, M. C., &Josephson, J. R. 1988. Some results concerning the computational complexity of abduction (Technical Report #88-TB-COMPLEXITY). Laboratory for Artificial Intelligence Research, Dept. of Cptr. & Information Science, The Ohio State Univ.

Eshelman, L. 1988. MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. In S. Marcus (Eds.), *Automating Knowledge Acquisition for Expert Systems* (37-80). Kluwer Academic Publishers.

Josephson, J., Chandrasekaran, B., Smith, J., &Tanner, M. 1987. A mechanism for forming composite explanatory hypotheses. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(3):445-454.

Josephson, J. R., &Goel, A. 1988. Tractable Abduction (Technical Report) The Ohio State University, Laboratory for Artificial Intelligence Research.

McDermott, J. 1988. Preliminary steps toward a taxonomy of problem-solving methods. In S. Marcus (Eds.), *Automating Knowledge Acquisition for Expert Systems* (225-256). Kluwer Academic Publishers.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge: Harvard University Press.

Punch III, W. F., Tanner, M. C., Josephson, J. R., &Smith, J. W. 1990. Peirce: A tool for experimenting with abduction. *IEEE Expert*, 5(5):34-44.