Determining what to learn through component-task modeling*

Bruce Krulwich
Center for Strategic Technology Research
Andersen Consulting LLP
100 S. Wacker Drive, Chicago, DL
krulwich@andersen.com

Larry Birnbaum Gregg Collins
The Institute for the Learning Sciences
Northwestern University
1890 Maple Ave, Evanston, IL
{birnbaum, collins}@ils.nwu.edu

ABSTRACT

Research in machine learning has typically addressed the problem of *how* and *when* to learn, and ignored the problem of formulating learning tasks in the first place. This paper addresses this issue in the context of the CASTLE system,[1] that dynamically formulates learning tasks for a given situation. Our approach utilizes an explicit model of the decision-making process to pinpoint which system component should be improved. CASTLE can then focus the learning process on the issues involved in improving the performance of the particular component.

1. Determining what to learn

A theory of learning must ultimately address three issues: when to learn, what to learn, and how to learn. The overwhelming majority of research in machine learning has been concerned exclusively with the last of these questions, how to learn. This ranges from work in purely inductive category formation to more knowledge-based approaches. The aim of this work has generally been to develop and explore algorithms for generalizing or specializing category definitions. The nature of the categories being defined—i.e., what is being learned—is rarely a consideration in the development of these algorithms. For purely inductive approaches, this is entirely a matter of the empirical data that serves as input to the learner. In explanation-based approaches (EBL), it is a matter of the user-defined "goal concept"—in other words, input of another sort. In neither case is the formulation of the learning task itself taken to be within the purview of the model under development.

Some work—in particular that in which learning has been addressed within the context of performing a task—has addressed the first question above, namely, when to learn. A common approach to this issue, known as *failure-driven learning,* is based on the idea that a system should learn in response to performance failures. The direct connection this establishes between learning and task performance has made this approach among the most widespread in learning to plan (e.g., Sussman, 1975; Schank, 1982; Minton, 1988; Hammond, 1989]. For the most part, however, even these models do not address the second question above, what to learn. In many cases, this is because the models are only capable of learning one type of lesson. What to learn is thus predetermined.

For example, many systems that learn exclusively from planner success always learn the same thing, namely a generalized form of the plan that was created (e.g., [Mitchell, 1990]). Similarly, many systems that learn from plan outcomes always learn the same type of planning knowledge--e.g., when it is feasible to make certain simplifying assumptions or to defer planning—from each situation (e.g., (Chien, 1990; DeJong *el al,* 1993]). Even systems that can learn more than one thing generally do so in a predetermined and inflexible fashion (e.g., [Hammond, 1989]).

While this type of solution can often be effective for any individual application setting, it fails to provide an account of how a learning system could determine for *itself* what to learn, and do so in a manner that is flexible enough to take account of the internal and external context of learning. For a system that is capable of learning a wide variety of types of concepts, in a wide variety of settings, the number of hard-wired mapping rules required to do this would be very large, and the rules themselves would get very difficult to manage or reason about, and may even be impossible to formulate.

More importantly, however, is the fact that hard-wired rules of this type do not provide a *theory* of determining what to learn. Just as a set of rules for actions can result in intelligent behavior without providing a foundation for the actions, hard-wired rules for determining what to learn can be effective but nonetheless do not necessarily provide a theory underlying these decisions. The point is that just as complex decisions about actions are very difficult to formulate using hard-wired rules, and thus require inference, so too complex decisions about what to learn require inference.

2. An everyday example

Consider the case of a person cooking rice pilaf for the first time. The last step in the directions says to "cover the pot and cook for 25-30 minutes." Suppose the person starts the rice cooking and then goes off to do something else—say,

The research presented here was carried out at The Institute for the Learning Sciences at Northwestern University, and is discussed in detail in the first author's Ph.D. thesis [Krulwich, 1993].
'CASTLE stands for Concocting Abstract Strategies Through Learning from Expectation-failures.

clean up the house. In the interim, the pot boils over. When the person returns to the kitchen a half-hour later, the rice pilaf is ruined.

What should be learned from this sequence of events? Intuitively we can imagine a number of lessons that might be learned:

- When a covered pot containing liquid is on the stove, keep an ear peeled for the sound of the lid bouncing or the sound of the water bubbling.
- Do not put a covered pot with liquid in it over a high flame, because it will boil over. The flame should be turned down.
- When cooking over a high flame, leave the pot uncovered or the lid ajar.
  Don't do loud things while cooking on the stove.
- When cooking liquid in a covered pot, stay in the kitchen.
- Don't cook over high flame when busy.

While all of these lessons are sensible, they are very disparate, in that they address very different issues. The lessons concern different aspects of behavior, refer to different portions of the agent's plan, and are expressed in different vocabularies. It is difficult to imagine how any learning process that did not distinguish among these alternatives in some way would be capable of such diverse behavior. Rather, it seems more likely that before the agent can undertake the task of learning from the mistake, he must select a lesson (or set of lessons) to learn. In other words, given that the agent has decided to learn from the mistake, and given that he is capable of carrying out the learning task, he still has to first determine what to learn.

We see, then, that the agent could learn several things in response to the rice pilaf boiling over. Which of the lessons the agent should learn, whether changes to the cooking methods, the idea of staying in the kitchen, or of tuning his perceptual apparatus, depend on the agent's perceptual and planning abilities, and on his knowledge of the domain. The key point is that many different lessons are possible. Any approach to determining what to learn must be flexible enough to account for this diversity.

3. Modeling cognitive tasks

What would an appropriate theory of determining what to learn look like? Imagine the thought processes going on in the agent's head (consciously or subconsciously) in viewing the situation and considering what lesson to learn:

Question 1: Why was the rice ruined?
Answer: The rice boiled over.
Question 2: Could I have done something differently at the time I started the rice cooking to prevent the problem ?
Answer: Yes, I could have lowered the flame or uncovered the pot.
Question 3: Without doing this, could I have prevented it from boiling over?
Answer: Yes, if I had heard it.
Question 4: Could I have heard it boiling over?
Answer: Maybe I could have, if I'd paid more attention.
Question 5: Why couldn't I hear it boiling over?
Answer: I was using the vacuum in the living room.
Question 6: Could I have planned things differently to enable me to hear?
Answer: Yes, I could have delayed vacuuming or stopped every few minutes to check the rice.
Question 7: Why didn't I?
Answer: I didn't think about the inability to hear from the other room.

The focus of this dialogue is on the decisions and actions of the agent that led to the rice burning, and particularly on what the agent could have considered or done to prevent the problem from arising. A self-dialogue of this sort is a means of analyzing the situation to explain what happened, and thereby focus learning from the experience [Chi *et. al,* 1989; Ram, 1989; Oehlmann *et. al,* 1993]. In this case the agent is performing *self-diagnosis,* trying to determine what mistake(s) he made that led to the rice burning. Put another way, the agent is considering possible lessons, and trying to determine which of them to learn. It is important to notice at this point that the dialogue is *not* specifically aimed at diagnosing the actions that the agent took to determine which action is to blame (although that may be involved as well). Rather, the dialogue is diagnosing the *decisions* that the agent made [Birnbaum *et. al,* 1990; Collins *et al,* 1993].

The key insight that will allow us to model this learning process is that each of the lessons that our agent can learn, and each of the questions in the hypothetical dialogue above, relates to a particular *cognitive task* that the agent was carrying out in the example:

- Listen harder for bubbling: *Perceptual tuning*
- Adjust the flame more carefully: *Plan step elaboration*
- Leave the lid ajar: *Plan step elaboration*
- Don't do loud things while cooking: *Scheduling interleaving*
- Stay in the kitchen while cooking: *Scheduling/ interleaving*
- Don't cook over high flame when busy: *Scheduling/ interleaving*

By "cognitive tasks" we mean here the classes of decisions that the agent makes in the course of decision-making. In our example, these cognitive tasks include such things as *plan step elaboration* (e.g., such as deciding how high to adjust the flame on the stove) and *perceptual tuning* (e.g., deciding what to listen for, in this case the sounds of the rice bubbling over). These tasks are themselves general cognitive abilities that are used frequently in goal-based behavior.

Given this insight, we can reformulate the learning problem posed above. Determining lessons to learn from a problem means first determining which cognitive tasks are relevant, and then determining what can be learned from the experience about how those particular tasks can be better carried out. In other words, we have transformed the problem of determining what to learn to two subproblems: determining what task to repair, and determining what aspect of the situation relates to that task [Krulwich, 1991, 1993].

4. Modeling planner structure

How can a computer system reason about its own decision-making and the cognitive tasks involved in that decision-making? The approach we will take is to design the system to facilitate this reasoning, by structuring its architecture in terms of *components* that carry out specific cognitive tasks [Collins *et. al,* 19911. In other words, we partition the system into chunks, each of which is responsible for a particular cognitive task, and treat each chunk as a component of the architecture. The behavior of the system, and the opportunities to improve it, can then be analyzed in terms of the behavior of components and the interactions between them.[2]

The next step, which answers the question of what to learn, is to associate with each component information about its ideal (desired) behavior. The system can use this information to determine what can be learned from the situation about better carrying out the component's task. The perceptual tuning component, for example, will have an associated description of the task of adjusting the agent's perceptual apparatus in response to its goals and environment. In the rice pilaf example the system could realize, based on this information, that it can learn a lesson about attending to the rice while cleaning the living room. If this kind of information is provided for each component, the system can determine what to learn by retrieving the relevant information for each component potentially in need of repair.

Given this component-based approach to modeling the planning process, how can a computer system diagnose which component is responsible for a failure? The process of self-diagnosis, as discussed above, is aimed not at diagnosing the agent's actions (at least not directly), but rather at diagnosing the decision-making constructs that gave rise to these actions. In other words, the over-arching question is not "What action of mine led to the problem?" but is rather "What deficiency in the way I make decisions led to the problem?" In contrast, previous research in learning to plan or solve problems in response to failures has generally been aimed at the first question, and has thus employed knowledge of the causal relations between the steps in the plan and the desired outcomes of those steps in diagnosis. When a plan failure occurs, this information is used to see what step in the plan resulted in the failure.

To diagnose failures in terms of faulty planner components, however, an agent requires analogous information concerning the causal relations between the decision-making processes used in planning, the actions taken, and the expected results. In other words, diagnosing the failure of a plan (or, more generally, of an expectation about the plan's performance) in terms of decision-making constructs requires information about the causal relations between the two. More concretely, the agent must reason explicitly about his *justification* for his actions in terms of his own reasoning mechanisms [Birnbaum *et. al,* 1990]. We consider our agent to know, or to be able to reconstruct, the reasons that he thought his decision-making was sound, and how his decision-making mechanisms led to the failure. Introspective dialogues such as the one we saw above correspond to the agent's examination of this justification, and his consideration of where the faults lie.

5. Flexible learning in the CASTLE system

We have seen that an intelligent agent must be able to dynamically determine what to learn, and that this process can require a significant amount of inference. By viewing the planning process as being composed of a variety of cognitive tasks, we transformed the problem of determining what to learn into two sub-problems: determining what cognitive task is at fault, and determining what could be learned to improve that task. What we still need to specify is how this process is initiated in the first place. Our approach is for the system to maintain and monitor explicit *expectations* that reflect the assumptions made during planning [Schank, 1982; Doyle *et. al,* 1986; Ortony and Partridge, 1987]. These expectations carry with them justification structures that relate them to the decision-making processes and otherwise-implicit assumptions that underly their being expected in the first place [Birnbaum *et. al,* 1990]. The failure of an expectation thus can directly lead to diagnosis of the relevant portions of the system's planning architecture.

This leads us to a fairly straightforward learning process. The planner considers the current situation and the active goals, and outputs a plan, along with a set of expectations to monitor. The failure of one of these expectations leads to diagnosis, which uses associated justification structures that represent part of the system's explicit self-model. The diagnosis process concludes which component is at fault, and how it should in fact have behaved. This information is passed to the repair module, which uses a model of the component's cognitive task to construct a repair.

We have implemented this approach in a system called CASTLE [Krulwich, 1991, 1993], CASTLE operates in the domain of chess, and learns new rules for a variety of cognitive tasks.

Consider the example shown in figure 1. In board (a) the opponent (playing white) chooses to move the queen to the

[2]This approach is often taken in reasoning about physical devices [Davis, 1990, sec. 7.1].

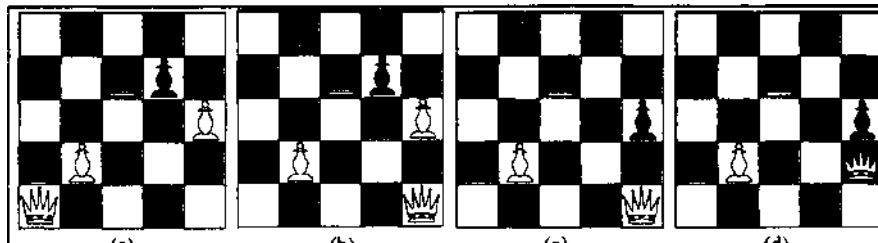

Figure 1: *Pin* example: Opponent (white) to move

right, to a square from which it can subsequently be moved to pin the computer's rook against the king. The computer (playing black) doesn't detect any strategies (offensive or option-limiting) that the opponent can execute, so it goes ahead with its own plan to capture the opponent's pawn. In board (c) the opponent moves its queen to pin the computer's rook, and the computer finds itself in board (d) with its rook pinned and a pawn (or the queen itself) able to make the capture in two moves.

The expectation that failed in this example relates to an assumption that is implicitly made by all intentional systems, that in general there will exist reasonable options—plans—that can be carried out for any goal that arises. This assumption underlies CASTLE's belief that it will be able to achieve its goals over the course of the game. One instantiation of this assumption, which CASTLE can directly monitor, is that the natural prerequisites to carrying out the plan, namely the ability to move pieces, will in general be met. In other words, the system's pieces will have mobility. Since the opponent presumably would like to limit the system's options, CASTLE uses its option-limiting planning rules during the plan recognition phase to check whether the opponent has the ability to limit CASTLE's options. If so, CASTLE will try to counterplan. If not, CASTLE will assume that its pieces will remain mobile until the subsequent turn. This process serves two purposes. First, any plans of the opponent's to limit CASTLE's options will hopefully be anticipated and countered. Second, if CASTLE fails to detect an opportunity for the opponent to limit the system's options, and the opponent takes advantage of the opportunity, CASTLE could learn a new option limiting planning rule in response to the failure. This is exactly what happens in the pin example.

This expectation failure invokes CASTLE's diagnosis and repair mechanisms. The diagnosis engine traverses the justification for the expectation, shown in figure 2, and determines that there must have been a plan executed by the opponent to limit the computer's options that was not generated by a method in the option-limit planning component. This fault description is then passed to CASTLE's repair module.

Once CASTLE has isolated the fault as a lack of an option-limiting planning rule, the repair mechanism takes over and first generates an explanation of why the opponent's move constituted an option-limited plan. To do this, the system retrieves a *component specification* that represents the purpose of the component being repaired. An explicit model of planning and execution is used to construct an explanation of how the component should have behaved in the example.

The component specification for the option-limiting planning component says roughly that an option-limiting plan is a single move that disables more than two opponent moves by the same piece for the same reason. This specification is used to generate the explanation shown in figure 3. The explanatory model specifies that a precondition for a move is that there is no opponent piece with a possible king threat through the square being vacated. The effect of the queen move, that the queen is now at its new location, conflicts with this precondition, because the queen in its new location does in fact have a possible king threat through the rook's location. This conflict explains why the queen's move disables six previously possible rook moves.

The learned rule for the pin as an option-limiting plan is shown in figure 4. The rule says roughly: *To compute a plan to limit the opponent's options, determine a piece to move, and a location to move to, and an opponent piece, such that the moved piece can attack the king with a move blocked only by the opponent piece.* This rule correctly predicts the opponent's ability to pin the computer's rook in the situation in figure 1(a), and can also be used offensively by CASTLE's planner to devise plans to limit the opponent's options.

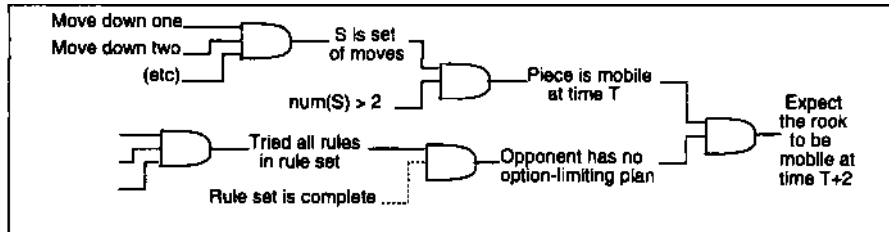

Figure 2: Justification of the mobility expectation in the *pin* example

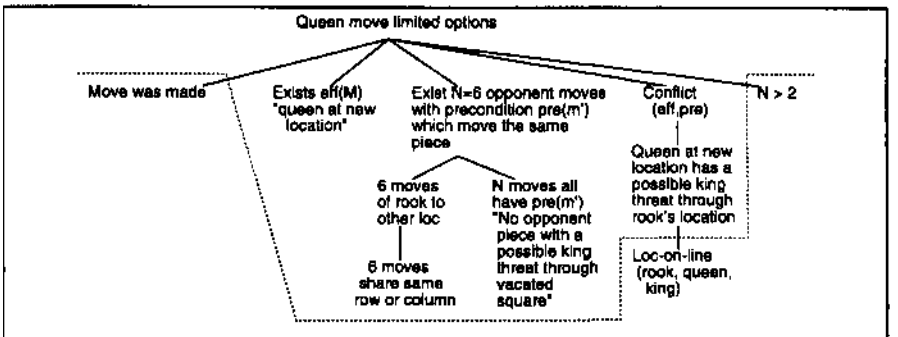

Figure 3: Explanation for the pin being an option-limiting move

6. Related work

How do other programs determine what to learn? In most cases there is only one type of concept to learn, so the determination is made in a fixed way by the program's control structure. CBG [Minton, 1984], for instance, learns a forced-move sequence schema whenever it loses a game. Theo-Agent [Mitchell, 1990] similarly learns a stimulus-response rule whenever its planner produces a new plan.

CHEF [Hammond, 1989] uses a fixed scheme of this sort to learn more than one type of construct. Whenever plan transformation is complete, CHEF stores the resulting plan back into its case library. Whenever a bug is found during simulation that was not anticipated earlier, the system learns a bug anticipator rule. Finally, whenever a bug is repaired, the system learns a generalized bug repair rule. The first thing to note is that the three types of things that CHEF can learn correspond to three components of a case-based planner, namely case retrieval, indexing, and adaptation (respectively). CHEF's relatively simple approach to the task of determining which of these three things to learn is highly effective for a number of reasons. First, there are only three types of things to learn, and they are highly distinctive. There is not much chance of confusing which category is applicable, as there might be if more subtle distinctions (e.g., between different types of indices relevant to different aspects of the planning process) were considered. Second, CHEF's plan simulator returns a complete causal analysis of the bugs that arise, so there is no need for complex inference to determine which of its components needs to be repaired, as there might be if CHEF were to learn in response to problems that arose later in time during plan execution. If either of these two conditions did not hold, a more complex approach would probably be needed.

SOAR (e.g., Newell, 1990]) uses a similar scheme to learn chunks (production rules) after resolving impasses. In SOAR's case, however, the chunks fit a variety of purposes because they are learned in response to impasses in different stages of decision-making. SOAR's impasses thus serve the same purpose as CASTLE's expectation failures. In the absence of explicit reasoning about what to learn, SOAR relies on its knowledge representations to direct the learning of new rules. In other words, decisions about when and what

```
(BRULE learned-limiting-plan-method3105
 (limiting-plan learned-limiting-plan-method3105 ?opp (world-at-time ?time)                 To limit the options
(plan (move ?opp move ?other-piece2  (loc ?r2 ?c2) (loc ?att-r ?att-c)) done)
    (goal-limit ?def-piece (loc ?def-r ?def-c)                                             of a defending piece
            (move (move ?opp move ?other-piece2 (loc ?r2 ?c2) (loc ?att-r ?att-c)))))
<=
(and (= ?def (player-opponent ?opp)) (= (current-game) ?chess)
     (at-loc ?opp ?other-piece2 (loc ?r2 ?c2) ?time)                                       Find a piece
     (move-legal-in-world (move ?opp move ?other-piece2 (loc ?r2 ?c2) (loc ?att-r ?att-c)) )  that can move to a new location
     (not (at-loc ?other-player ?other-piece (loc ?att-r ?att-c) ?time))                   that's currently empty
     (at-loc ?def ?def-piece (loc ?def-r ?def-c) ?time)                                    and a defending piece
     (move-legal-in-world (move ?def move ?def-piece  (loc ?def-r ?def-c) (rc->loc ?r ?c)) )  that can move
     (at-loc ?def king (loc ?to-row ?to-col) ?time)                                        such that the piece
     (move-legal (move ?opp (capture king) ?other-piece2                                   can legally attack
                 (loc ?att-r ?att-c) (loc ?to-row ?to-col)))                               the opponent's king
     (newest-loc-on-line (loc ?def-r ?def-c) (loc ?att-r ?att-c) (loc ?to-row ?to-col))    but is blocked by the defending piece
     (no (and (newest-loc-on-line (loc ?other-r ?other-c)                                  and such that no other square
                          (loc ?att-r ?att-c) (loc ?to-row ?to-col))                       on the line of attack
          (not (= ?other-r ?def-r)) (not (= ?other-c ?def-c))                             is occupied
          (at-loc ?other-player2 ?other-piece3 (loc ?other-r ?other-c) ?time)))))
```

Figure 4: Learned rule for limiting options by pinning

to learn are tied directly to the representation of problems and sub-problems, and thus to the expressiveness of the system's vocabulary. The lack of explicit control over learning often leads to the creation of undesirable chunks, and the only solution within the SOAR philosophy has been to limit the expressiveness of the system's vocabulary [Tambe and Rosenbloom, 1988]. In contrast, CASTLE's approach is to reason explicitly about what rules should be learned, in a sense taking the opposite approach of *extending* the system's vocabulary. It may be possible to extend SOAR to carry out explicit reasoning of this sort as another heuristic search problem, but this has not been investigated.[3]

PRODIGY [Minton, 1988] is probably the closest in spirit to CASTLE, in that it performs dynamic inference to determine what to learn in a given situation. The system has a set of *example recognizers* that look for conditions indicating that a node in a problem-solving trace is an instance of a particular type of concept. For example, if a node refers to a choice of an operator to apply, and the choice ended up failing, an example recognizer will signal the node as an example of an operator that should not have been chosen. This node, along with the concept it is believed to be an instance of, is passed to an EBL mechanism which constructs a new search control rule.

This process appears to satisfy the criteria discussed above for analytical approaches to determining what to learn. Moreover, this "pattern-matching" approach to determining what to learn seems quite appealing, because it appears to require fairly little inference (certainly less than CASTLE's diagnostic approach), and because it works using clear declarative knowledge about each of its types of rules.

In practice, however, assessment of PRODIGY'S approach is more complicated, in part because the system additionally requires the use of procedurally encoded *example selection heuristics* to "eliminate uninteresting examples" [Minton, 1988, sec. 4.2]. While the rationale to these heuristics sounds innocuous, they in fact embody quite sophisticated reasoning. Some of these rules select interesting examples for particular rule types (one per rule type), others are used to search for specific conditions that make learning a particular type of rule *beneficial,* and others are used for determining "interestingness"of a number of rule types. All of these are separate from the initial example recognition process. Additionally, these functions maintain a history of the process of searching for examples to learn that is used in subsequent example selection.

All in all, it seems clear that PRODIGY's determination of what to learn is far more complex than the simple recognition of patterns in the problem-solving trace. The process employs a great deal of heuristic information about what will constitute a good search control rule, and this information enables PRODIGY to learn effectively. Were it not for these heuristics, the learning process would spend an inordinate amount of time learning pointless search control rules.[4]

The point, however, is not that PRODIGY is in any way wrong to carry out complex reasoning of this sort to determine what to learn—indeed, the fundamental claim of this paper is that such inference is necessary. Rather, the point is that while PRODIGY relegates the complexities of example selection to procedurally-encoded heuristics, the research presented in this paper attempts to make such reasoning and information explicit.

A number of recent research endeavors have taken an approach similar to CASTLE's in carrying out explicit inference to determine how to apply learning routines to a given situation. One significant initiative is in the area of *multi-strategy learning* [Michalski, 1993], in which systems have the ability to apply a number of learning algorithms to a problem, and use dynamic inference to determine which is best. This work is certainly similar in spirit to CASTLE, but it is important to note that it does not inherently address the issue of determining what to learn, rather it represents an inferential method of determining how to learn.

A number of other research projects have addressed more specifically the issue of *learning goals* [Hunter, 1989; Cox and Ram, 1992; Leake and Ram, 1993; Michalski, 1993] The use of learning goals *per se* does not imply inferential determiniation of what to learn, because such goals are often treated simply as inputs to the system. Some of these projects, however, decide dynamically which learning goals to pursue, and as such directly address the issues we have been discussing in this paper. Several of these projects take approaches that are strikingly similar to CASTLE's, notably in the areas of story understanding [Cox and Ram, 1992] and case-based planning [Oehlmann *et. al,* 1993; Fox and Leake, 1994].

7. Conclusions

CASTLE's approach to modeling planning in terms of semantically meaningful components gives it the ability to reason dynamically about what to learn. CASTLE is currently able to learn twelve strategies, including forking, pinning, and boxing in, that relate to a variety of cognitive tasks [Krulwich, 1993]. This is possible because of the system's ability to reason explicitly about its tasks and subtasks. Previous research has determined what to learn in a fixed way, either by wiring the determination into the control structure of the program, or by providing complex *ad hoc* methods for making the determination.

The guiding theme throughout this research has been the use of self-knowledge in learning to plan [Collins *et. al,*

[3]This appraoch would seem consistent with SOAR's methodology of expressing all aspects of problem-solving in terms of searchbut it would raise the question of the source of SOAR's leverage, its architecture or its representational models. For this reason the approach might be said to undercut SOAR's goal of achieving intelligent behavior through a uniform architecture.

4 Thanks to Steve Minton for personal communication clarifying the issues involved in PRODIGY's selection heuristics.

1993]. Several forms of self-knowledge have been delineated, including component specifications, justifications, and explanatory models, and learning algorithms have been developed and adapted to use this knowledge properly. The reification of this knowledge and methods for using it effectively form the bulk of the contributions made by the research.

The CASTLE system, while demonstrating the viability of the approach, is only a first step in implementing a system that learns using self-knowledge, and there are many areas of open research in extending the application of these ideas. One such area is to apply the approach to more complex decision-making methods, such as non-linear planning, case-based reasoning, or hierarchical planning, in which the approach consists of repeated application of a number of sub-processes, each of which would be explicitly modeled. Another area is in more complex domains of application, such as robotic planning, complex route planning, or scheduling, which would again require extending the system's models.

In a broader sense, the research suggests an agenda exploring the use of self-models in learning, planning, and understanding. The formulations of self-knowledge that are useful in learning should also give leverage into planning, execution, knowledge acquisition, communication, understanding, and design.

Acknowledgements

The research described in this paper was done in collaboration with Mike Freed. The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting. The Institute receives additional support from Ameritech and North West Water, corporate sponsors.

References

Birnbaum, L., Collins, G., Freed, M., and Krulwich, B., 1990. Model-based diagnosis of planning failures. In *Proceedings of the 1990 National Conference on Artificial Intelligence.*

Chi, M., Bassok, M., Lewis, M., Reimann, P., and Glaser, R., 1989. Self-explanations: How students study and use examples to solve problems. *Cognitive Science,* 13:145-182.

Chien, S., 1990. *An explanation-based learning approach to incremental planning.* Ph.D. thesis, University of Illinois at Urbanna-Champaign.

Collins, G., Birnbaum, L., Krulwich, B., and Freed, M. 1991. Plan debugging an intentional system. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence.*

Collins, G., Birnbaum, L, Krulwich, B., and Freed, M., 1993. The role of self-models in learning to plan. In *Foundations of Knowledge Acquisition: Machine Learning,* pages 83-116. Kluwer Press, Boston, MA. (Also technical report #24, The Institute for the Learning Sciences, 1992.)

Cox, M., and Ram, A., 1992. Multistrategy learning with introspective meta-explanations. In *Proceedings of the 1992 Machine Learning Conference.*

Davis, R., 1990. *Representations of commonsense knowledge.* Morgan Kauffman, San Mateo, CA.

DeJong, G., Gervasio, M., and Bennett, S., 1993. On integrating machine learning with planning. In *Foundations of Knowledge Acquisition: Machine Learning,* pages 83-116. Kluwer Press.

Doyle, R., Atkinson, D., and Doshi, R., 1986. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the 1986 National Conference on Artificial Intelligence,* pages 81-87.

Fox, S. and Leake, D., 1994. Using introspective reasoning to guide index refinement in case-based reasoning. In *Proceedings of the 1994 Conference of the Cognitive Science Society.*

Hammond, K., 1989. *Case-based planning: Viewing planning as a memory task.* Academic Press, San Diego, CA. Also Yale technical report #488.

Hunter, L., 1989. *Knowledge-acquisition planning: Gaining expertise through experience.* Ph.D. thesis, Yale University.

Krulwich, B., 1991. Determining what to learn in a multi-component planning system. In *Proceedings of the 1991 Cognitive Science Conference,* pages 102-107.

Krulwich, B., 1993. *Flexible learning in a multi-component planning system.* Ph.D. thesis, The Institute for the Learning Sciences, Northwestern University. Technical report #46.

Leake, D., and Ram, A., 1993. Goal-driven learning: Fundamental issues and symposium report. Tech. report 85, Indiana University.

Michalski, R., 1993. *Machine Learning,* special issue on Multistrategy Learning, 11(2/3).

Minton, S., 1984. Constraint-based generalization: Learning game-playing plans from single examples. In *Proceedings of the 1984 National Conference on Artificial Intelligence.*

Minton, S., 1988. *Learning effective search-control knowledge: An explanation-based approach.* Ph.D. thesis. School of Computer Science, Carnegie Mellon University. Technical report CMU-CS-88-133. Also published by Kluwer Academic Publishers.

Mitchell, T., 1990. Becomming increasingly reactive. In *Proceedings of the National Conference on Artificial Intelligence.*

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

Ochlmann, R., Sleeman, D., and Edwards, P., 1993. Learning plan transformations from self-questions: A memory-based approach. In *Proceedings of the 1993 National Conference on Artificial Intelligence,* pp. 520-525.

Ortony, A., and Partridge, D., 1987. Surprisingness and expectation failure: What's the difference? In *Proceedings of the 1987 International Joint Conference on Artificial Intelligence.*

Ram, A., 1989. *Question-driven understanding: An integrated theory of story understanding, memory, and learning.* Ph.D. thesis, Yale University.

Schank, R., 1982. *Dynamic Memory.* Cambridge University Press, Cambridge, England.

Sussman, G., 1975. *A Computer Model of Skill Acquisition.*

Tambe, M., and Rosenbloom, P., 1988. Eliminating expensive chunks. Technical report CMU-CS-88-189, School of Computer Science, Carnegie Mellon University.