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

Title
Learning from Failures for Cognitive Flexibility

Permalink
https://escholarship.org/uc/item/3kb9899n

Journal
Proceedings of the Annual Meeting of the Cognitive Science Society, 32(32)

ISSN
1069-7977

Authors
Choi, Dongkyu
Ohlsson, Stellan

Publication Date
2010

Peer reviewed

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

Learning from Failures for Cognitive Flexibility

Dongkyu Choi (dongkyuc@uic.edu)
Stellan Ohlsson (stellan@uic.edu)
Department of Psychology
University of Illinois at Chicago
1007 W Harrison Street (M/C 285), Chicago, IL 60607 USA

Abstract

Cognitive flexibility is an important goal in the computational modeling of higher cognition. An agent operating in the world that changes over time should adapt to the changes and update its knowledge according to them. In this paper, we report our progress on implementing a constraint-based mechanism for learning from failures in a cognitive architecture, ICARUS. We review relevant features of the architecture, and describe the learning mechanism in detail. We also discuss the challenges encountered during the implementation and describe how we solved them. We then provide some experimental observations and conclude after a discussion on related and future work.

Keywords: cognitive architecture, constraints, constraint violations, learning from failures, skill acquisition

Introduction

In computational models of higher cognition, it is important to simulate the broad human functionality that we call *adaptability* or *flexibility*. Cognitive flexibility is, of course, a multi-dimensional construct, but in this paper, we focus specifically on the ability of humans to act effectively when a familiar task environment is changing, thus rendering previously learned skills ineffective or obsolete.

Traditionally, researchers discussed two types of error correction mechanisms for this problem. *Weakening* (Anderson, 1983, pp. 249–254) assumes that certain knowledge structures like rules, skills, schemas, or chunks have strengths associated with them, and it decreases the strength of the particular structure that generates a negative outcome. However, actions themselves are not typically correct or incorrect, or appropriate or inappropriate. Instead, they are appropriate, correct or, useful in some situations but not in others. The goal of learning from failure is thus to distinguish between the class of situations in which a particular type of action will cause errors and the class of situations in which it does not. Weakening does not accomplish this, because lower strength makes an action less likely to be selected in all situations.

Another mechanism proposed for error correction is *discrimination* (Langley, 1987). The key idea behind this contribution is to compare a situation with a positive outcome and another with a negative outcome to identify discriminating features. If an action generates both positive and negative outcomes across multiple situations, the system identifies any features that were true in one situation but not in the other, and uses them to constrain the applicability of the action. But the computational discrimination mechanism also has several problems including: the lack of criterion for how many instances of either type are needed before a valid inference as to the discriminating features can be drawn; the possible existence of a very large number of potential discriminating features, leading to complex applicability conditions or large numbers of new rules or both; and the inability to identify potential discriminating features with a causal impact from those of accidental correlation.

In response, Ohlsson (1996) developed a *constraint-based specialization* mechanism for learning from negative outcomes. The production system implementation of the mechanism overcomes most of the weaknesses of previous methods. It assumes that the agent has access to some declarative knowledge in the form of *constraints*, which consist of an ordered pair with a relevance criterion and a satisfaction criterion. The system matches the relevance criteria of all constraints against the current state of the world on each cycle of its operation. For constraints with matching relevance conditions, the system also matches the satisfaction conditions. Satisfied constraints require no response, but violated constraints signal a failed expectation due to various reasons including a change in the world or erroneous knowledge. This constitutes a learning opportunity, and the system revises the current skill in such a way as to avoid violating the same constraint in the future. The computational problem involved here is to specify exactly how to revise the relevant skill when an error occurs, and the constraint-based specialization provides a solution to this problem.

Unlike weakening, the constraint-based approach identifies the specific class of situations in which an action is likely (or unlikely) to cause errors. It also differs from the discrimination method, and the mechanism does not carry out an uncertain, inductive inference. Instead, it computes a rationally motivated revision to the current skill. However, these advantages were limited by a simplistic credit/blame attribution algorithm and the lack of serious architectural supports like other learning mechanisms that can operate in parallel. In this paper, we adapt the constraint-based specialization mechanism to a cognitive architecture, ICARUS, and address these limitations. The architecture features hierarchical knowledge structures, and it has a variety of well-developed capabilities including learning from positive outcomes (Langley & Choi, 2006). We first review the relevant features of the ICARUS architecture, and describe the constraint-based specialization mechanism in some detail. Then we identify the challenges we encountered during the implementation in ICARUS, with a particular attention to the credit assignment problem. Finally, we report some experimental observations with the system, and discuss related and future work.

The ICARUS Architecture

Cognitive architectures aim for a general framework for cognition. They include a set of hypotheses covering representation, inference, execution, learning and other aspects of cognition. Soar (Laird et al., 1986) and ACT-R (Anderson, 1993) are some of the well-known cognitive architectures, and the ICARUS architecture exhibits some similarities to them but has some important differences as well (Langley & Choi, 2006). In this section, we review the fundamental aspects of the architecture before continuing our discussion to the specifics of learning from failures in this framework.

Representation and Memories

ICARUS distinguishes conceptual and procedural knowledge. Concepts describe the environment, and enable the system to infer beliefs about the current state of the world. Skills, on the other hand, consist of procedures that are known to achieve certain goals. The architecture also distinguishes long-term, abstract knowledge and short-term, instantiated structures. Long-term concepts and skills are general descriptions of situations and procedures, and the system instantiates them before applying them to a particular situation. Instantiated concepts and skills are short-term structures, in that they are applicable only at a specific moment. ICARUS has four separate memories to support these distinctions.

The architecture encodes concepts with definitions that are similar to Horn clauses. They consist of a head and a body that includes perceptual matching conditions or references to other concepts. Table 1 shows some sample concepts. The first concept has a head, `(same-color ?block1 ?block2)`, and specifies perceptual matching conditions among the variables involved in its `:percepts` and `:tests` fields. It is a *primitive* concept, which does not have any reference to other concepts. The second concept also has a head and some perceptual matching conditions, but it has references to other concepts in the `:relations` field, and therefore, it is a *non-primitive* concept.

Table 1: Some sample ICARUS concepts for the Blocks World. Question marks denote variables.

```
((same-color ?block1 ?block2)
 :percepts  ((block ?block1 color ?color)
             (block ?block2 color ?color))
 :tests     ((not (equal ?block1 ?block2))))

((not-color-sorted ?color)
 :percepts  ((block ?block1 color ?color)
             (block ?block2))
 :relations ((on ?block1 ?block2)
             (not
              (same-color ?block1 ?block2))))
```

On the other hand, ICARUS' skills resemble STRIPS operators. The head of each skill is the predicate it is known to achieve, and therefore, all skills are indexed by their respective goals. Each skill has a body that consists of perceptual matching conditions, some preconditions, and either direct actions to the world or references to its subgoals. Like in concepts, skills with no references to any subgoals are *primitive*, while the ones with them are *non-primitive*. The hierarchical organization provides multiple layers of abstraction in the specification of complex procedures.

In Table 2, the first skill indexed by its goal `(stacked ?block ?to)` has some perceptual matching conditions and a precondition, `(stackable ?block ?to)`. It includes several direct actions in the world (marked with asterisks), and therefore, it is a primitive skill. The second skill, however, is a non-primitive one, with references to subgoals, `(stackable ?block1 ?block2)` and `(stacked ?block1 ?block2)`. The subgoals are ordered, and they invoke other skills that achieve them. For instance, the second subgoal will invoke skills like the first example in the table. In this manner, ICARUS's skills are hierarchically organized.

Table 2: Some sample ICARUS skills for the Blocks World.

```
((stacked ?block ?to)
 :percepts ((block ?block)
            (block ?to xpos ?xpos ypos ?ypos
                       height ?height))
 :start    ((stackable ?block ?to))
 :actions  ((*horizontal-move ?block ?xpos)
            (*vertical-move ?block
                            (+ ?ypos ?height))
            (*ungrasp ?block)))

((on ?block1 ?block2)
 :percepts ((block ?block1)
            (block ?block2))
 :subgoals ((stackable ?block1 ?block2)
            (stacked ?block1 ?block2)))
```

Inference and Execution

The ICARUS architecture operates in cycles. On each cycle, the system instantiates its long-term concepts based on the current situation. The bottom-up inference of concepts creates beliefs in the form of instantiated conceptual predicates. The inference process starts with the perceptual information about objects in the world. The system attempts to match its concept definitions to the perceptual information and, when there is a match, it instantiates the head of the definitions to compute its current beliefs.

Once the architecture computes all its beliefs, it starts the skill retrieval and execution process. ICARUS' goals guide this process, and the system retrieves relevant long-term skills based on the current beliefs. When it finds an executable path through its skill hierarchy, from its goal at the top to actions at the bottom, ICARUS executes the actions specified at the leaf node of the path. This execution, in turn, changes the environment, and the system starts another cycle by inferring the updated beliefs from new data received from the environment.

Problem Solving and Learning

During the execution for its goals, the architecture sometimes encounters a situation where it can not find any executable skill path. When this happens, ICARUS invokes its means-ends problem solver, chaining backward from its goal. It at-

tempts to use two types of *chains*, a skill chain that uses a goal-achieving skill with unsatisfied preconditions and a concept chain that decomposes the goal into subgoals through concept definitions. Once the system finds a subgoal with an executable skill during this process, it immediately executes the skill and continue to the next cycle until it achieves all the top-level goals.

When the architecture finds a solution and achieves a goal (which includes both the top-level goals and any of their subgoals), it learns new skills from the successful problem solving trace. The learned skills differs in their forms based on the type of the problem solving chain. Further discussions on the problem solving and learning capabilities would require more space than we can afford here, but Langley and Choi (2006) covers all the details. In the subsequent sections, we explain the details of the constraint-based specialization mechanism and its implementation in ICARUS.

Learning from Failures

As described in the previous section, ICARUS has hierarchically organized skill knowledge and it can learn from positive outcomes through problem solving. However, the architecture can not adapt to environmental changes when some of its existing skills become incorrect or obsolete. Extending ICARUS with the constraint-based specialization mechanism provides this capability.

Representation of Constraints

The extended architecture stores each constraint as a pair of relevance and satisfaction conditions, following Ohlsson and Rees (1991). Both relevance and satisfaction conditions are conjunctions of predicates, and the ICARUS architecture keeps a list of such pairs in a separate constraint memory.

Table 3 shows some sample constraints we use in the Blocks World domain. For convenience, we store each pair with a name like *color*, *top-block*, or *width*. The first constraint, *color*, says that two blocks should have the same color when they are stacked, which, in effect, enforces all towers to have a single color. Similarly, the other two constraints mean that a block that is designated as a *top-block* should always be clear, and that a block on top of another block should be smaller than the one below, respectively.

Table 3: Some sample constraints for the Blocks World.

```
(color     :relevance    ((on ?a ?b))
           :satisfaction ((same-color ?a ?b)))
(top-block :relevance    ((top-block ?b))
           :satisfaction ((clear ?b)))
(width     :relevance    ((on ?a ?b))
           :satisfaction ((smaller-than ?a ?b)))
```

Detection of Constraint Violations

On each cycle, the system checks if the current belief state satisfies all the constraints. It first attempts to match the relevance conditions of its constraints against the current state, and, if a match is found, verifies that the satisfaction conditions also hold. When it finds an unsatisfied constraint, it attempts to revise the skill that caused this violation.

We distinguish two different types of constraint violations. In the first type, a constraint just becomes relevant after an action but not satisfied at the same time. For instance, when an agent stacks a red block, *A*, on top of a blue block, *B*, it achieves `(on A B)`, so the corresponding instance of the color constraint in Table 3 matches and the constraint becomes relevant by the stacking action. But the satisfaction condition, `(same-color A B)`, is not met in this case, because one of the blocks is red and the other is blue. We refer to violations like this as *type A* violations.

Another type of violations, which we call *type B* violations, involves a constraint that has been relevant and satisfied, but becomes unsatisfied as a result of an action while it still stays relevant. An example of this type occurs when an agent stacks a block *C* on top of a block *TB* that is designated as a top block. In this case, the top-block constraint stays relevant before and after the stacking action, since the predicate, `(top-block TB)` continues to hold. But the satisfaction condition, `(clear TB)` becomes false as a consequence of the action, and the constraint is violated.

Skill Revisions

Once the system detects constraint violations of either type, it randomly chooses one of them and attempts to make revisions to the skill it just used. The revision process shares its basic steps with those used in previous research (Ohlsson, 1996; Ohlsson & Rees, 1991). The goal of this process is to constrain the application of the skill to situations in which it will not violate the constraint.

For a type A violation, where a constraint becomes relevant but violated, one of the revisions forces the constraint to stay irrelevant, and the other ensures that it is both relevant and satisfied. On the other hand, a type B violation, in which a constraint stays relevant but becomes violated, invokes one revision that makes sure the constraint is irrelevant, and another that restricts the use of the skill to cases where the satisfaction is not affected.

The system revises skills by adding preconditions, and Table 4 shows how the system computes the new preconditions for the two types of violations. C_r and C_s represent the relevance and satisfaction conditions. O_a and O_d are the add and delete lists of the executed primitive skill. The rationale for these computations has been developed in detail in prior publications (Ohlsson, 1996; Ohlsson & Rees, 1991).

As an example, let us revisit the Blocks World cases. In the first case, we have a red block, *A*, and a blue block, *B*. The system executes an instance of the first skill shown in Table 2, `(stacked A B)`, which adds the predicate to the state. This implies that the relevance condition of the color constraint, `(on A B)` also becomes true, but the satisfaction condition `(same-color A B)` does not. When detecting this type A violation, the system computes additional preconditions and attempts to make two revisions. The first calculation, $\neg(C_r -$

Table 4: New preconditions created in response to constraint violations.

Type \ Revision	1	2
A	$\neg(C_r - O_a)$	$(C_r - O_a) \cup (C_s - O_a)$
B	$\neg C_r$	$C_r \cup \neg(C_s \cap O_d)$

O_a), leads to `(on A B) − (stacked A B)`, which results in a null precondition. Therefore, the system ignores the first revision and tries the second one. This time, the additional precondition comes from $(C_r - O_a) \cup (C_s - O_a)$, which leads to `(same-color A B)`. The system adds this precondition to the skill that caused the violation, and restricts the execution of the stacking action to the case where two blocks have the same color.

In the second case, we have two blocks, *C*, and *TB*. When the system stacks the block C on top of the block TB using the skill `(stacked C TB)`, the top-block constraint becomes unsatisfied (`(clear TB)` not true in the state) while it stays relevant continuously (`(top-block TB)` true in the state). Upon detecting this type B violation, the system computes two sets of additional preconditions using the formulas, $\neg C_r$ and $C_r \cup \neg(C_s \cap O_d)$. These lead to `(not (top-block TB))` and `((top-block TB) (not (clear TB)))`, respectively, which are added to two separate revisions of the skill. The first revision prevents the use of the stacking action onto a block designated as a top-block. The second revision is a case of over-specialization, which makes it impossible to fire. Nevertheless, the two revisions achieve the proper restriction of the skill for the top-block constraint.

Challenges in Implementation

Although this implementation in the context of ICARUS shares the basic steps with previous systems using constraint-based specialization, various important differences between the ICARUS architecture and production system architectures require some significant changes in the revision process. In this section, we discuss the challenges and our solutions to them.

Hierarchical Organization

First of all, ICARUS' hierarchical organization of skill knowledge poses the most significant change, in relation to the assignment of blame. Production systems have flat structures, and it is mostly the case that the last executed rule caused a violation. But in ICARUS, execution involves a skill path, which may include more than one skill instance. Skill instances near the top of the path are more abstract, and those close to the bottom are more specific. Depending on the level of abstraction at which the violated constraint exists, the skill that needs to be revised can be anywhere on this path, and no simple attribution rule will be sufficient. So, the question is how ICARUS can identify the right skill to revise generally.

An analysis of multiple examples indicates that the architecture should find the highest level in the skill path in which all the variables involved in the additional preconditions for the revision are bound. All the additional preconditions are fully instantiated at this level and, therefore, it is the highest level in which the preconditions become meaningful. This makes it the right level at which to make the corresponding revisions. The results of running ICARUS indicate that this solution is correct. This solution is easily computable and general across domains. The possibility that it applies to other types of hierarchical systems might deserve attention.

Add and Delete Lists

Another problem occurs during the computation of the additional preconditions for skill revisions. Unlike production systems that have explicit and complete add and delete lists associated with actions, the ICARUS architecture has skills associated with goals. Goals typically do not include any side effects we do not care about, and they do not specify any predicates that should disappear after a successful execution. For this reason, the add and delete lists are not explicit in the architecture, and we must compute them from other sources.

Meanwhile, the use of add lists during the revision process is limited to the calculation of logical differences, and we can use goals as if they represent complete add lists. This will make the revised skill more restrictive rather than less so, thus making it safe. However, we should compute the delete list explicitly because of the way it is used during the revision process. We chose to calculate the list by comparing two successive belief states, although this may include some predicates removed by sources external to the agent. Again, however, this makes the revisions more restrictive, rather than more general, keeping the agent safe, because the delete list is negated during the computation of preconditions.

Disjunctive Definitions

ICARUS' support for multiple, disjunctive definitions of concepts adds another layer of complexity. When computing additional preconditions for skill revisions, the system should decompose any non-primitive concepts. Disjunctive concepts create multiple expansions, possibly resulting in more than one set of additional preconditions. The architecture accepts all such expansions and create multiple revisions.

The consequences of this approach are significant. When the system experiences a constraint violation, the situation might involve a particular disjunction of a concept. Nevertheless, the architecture learns multiple revisions from this case, covering all possible disjunctions of the concept. This approach is based on the understanding that there is a good reason why the disjunctive concepts have the same head, and that the system benefits from learning about all such cases. In future tasks, the system might confront a situation in which another one of the disjunctions applies, and, due to its prior learning, the system will already know how to avoid making an error in this situation even though it has never encountered it before.

Experimental Observations

To verify that the system works as intended, we performed experiments in two domains. We give only the basic concept and skill sets to the system at the beginning, along with the information on constraints. This means that the system knows how to operate in the world, but not at the level of expertise that enables it to satisfy the constraints at all times. It is as if humans sometimes know what should happen and what should not, but do not necessarily know how to impose these rules and often make mistakes. As the system learns from its failures, it revises the basic skills to avoid constraint violations in the future.

Blocks World

We modified the familiar Blocks World from the typical setup to include blocks of different colors and sizes. This modified domain supports various constraints like the color and size of the blocks in a tower or the maximum height of each tower. Table 3 shown earlier includes three of the constraints we have in this domain. The *color* constraint says whenever a block is stacked on top of another the two blocks should have the same color. This, in effect, forces any tower to have only the blocks of one color. The *top-block* constraint means any block designated as a top block (according to the system's conceptual knowledge) should always be clear, having no other blocks on top. The last constraint, *width* enforces that a block is smaller than the block underneath it.

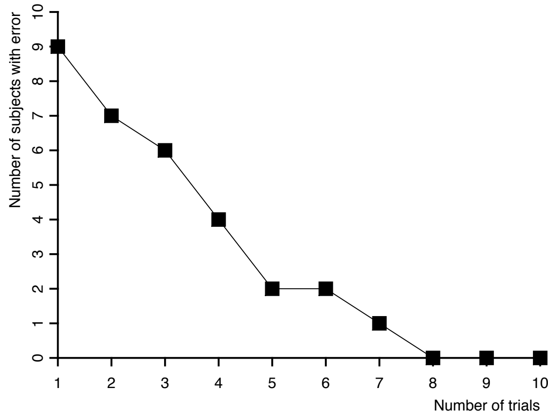

Figure 1: Number of simulated subjects that violated a constraint at each trial.

We ran simulation experiments with several different goals, and Figure 1 shows the result from one of them. In this experiment, we had ten simulated subjects, and each subject performed ten trials of the given task. We recorded the number of subjects that violated any constraints during each trial. The graph clearly shows that the revision process gradually reduces the number of the simulated subjects with constraint violations.

Route Generation

Another domain we used to test the system is a simplified version of route generation between places. Here, in addition to testing the specialization mechanism in ICARUS, we also want to verify that the mechanism can operate in parallel to other learning schemes like learning from success. The agent starts at a certain location, and has the goal of getting to a target location elsewhere. Using the information on connections between neighboring locations, the system performs problem solving to find a route to its target. As a result, it finds one of the several possible routes that involve different waypoints, and ICARUS learns specific route knowledge from this positive experience.

But some of the routes might become unavailable for travel due to various reasons like a broken bridge. At subsequent runs, the agent encounters situations where it can not use routes it learned before. While attempting to get to the target using a learned route, ICARUS recognizes that it gets stuck at a location with no outlet, violating a constraint not to be at a dead end. This failure triggers the system to learn a revised skill, which prevents it from moving to a location without any outlet. On the next trial, armed with this new skill, the system attempts to find another route to get to its target, and learns a skill for an alternate route for later use.

Let us see this behavior in a sample run. We give the system a goal to get to a target location, *B*, starting from the initial location, *A*. The two locations are connected by two alternate routes using waypoints *W1* and *W2*, respectively. The system starts out with two concepts and a skill as shown in Table 5. It also has the connection information between the locations, *A*, *B*, *W1*, and *W2* as some static beliefs. The only constraint it knows of is,

```
(at ?location) → (not-dead-end ?location)
```

which simply says that it should not be at a dead end at any time. During the first trial, the system finds a path, *A - W1 - B* through problem solving, and learns a specific skill for this route. Before we continue to the next trial, we intentionally remove the connection between *W1* and *B*, making the path obsolete. On the next trial, the system attempts to reuse the path, but it finds that it violates the constraint while it is at location *W1*. This violation triggers a revision process, resulting in another new skill. Once the system learns this new skill, it attempts to find an alternate route through yet another problem solving process, resulting in the path, *A - W2 - B*. After ICARUS stores this route as a specific skill, it executes the skill when it encounters the same task at a later time.

Related and Future Work

The current work on the constraint-based specialization has important similarities to some work in the explanation-based learning (EBL) literature (see Ellman, 1989; Wusteman, 1992 for reviews). EBL methods assume a significant amount of domain theories presumed to be perfect. However, in most of the domains, this is not true, and they require some ways

Table 5: Two concepts and a skill given to ICARUS for the route generation domain, and the two skills the system learned. The first skill is learned from problem solving (marked as P-S), and the other is learned from constraint-based specialization (marked as C-S). The additional precondition in this skill is shown in bold face.

Given:	((at ?location) :percepts ((self ?self location ?location))) ((not-dead-end ?location) :percepts ((location ?location)) :relations ((connected ?location ?to1) (connected ?location ?to2)) :tests ((not (equal ?to1 ?to2)))) ((at ?location) :percepts ((location ?from)) :start ((at ?from) (connected ?from ?location)) :actions ((*move-to ?location)))
Learned from P-S 1:	((at B) :subgoals ((at W1) (at B)))
Learned from C-S:	((at ?location) :percepts ((location ?from)) :start ((at ?from) (connected ?from ?location) **(not-dead-end ?location)**) :actions ((*move-to ?location)))
Learned from P-S 2:	((at B) :subgoals ((at W2) (at B)))

to augment or correct the domain theories. There, researchers worked on the similar problems of blame assignment and theory revision, although the exact formulations were different. Unlike most of these work, our approach uses explicit descriptions of constraints, which the system uses to detect failures and revise existing theories accordingly.

With the successful implementation of the constraint-based specialization mechanism in ICARUS, we are able to study the important problem of interactions between two learning mechanisms. People learn in a variety of ways (Ohlsson, 2008) and human-level flexibility is the outcome of the interactions among multiple learning mechanisms. Currently, we have only a limited understanding of how learning mechanisms interact to produce flexible behavior. We intend to add additional mechanisms to ICARUS, including learning from examples or from analogies, and explore the conditions under which multiple mechanisms produce more flexible behavior than individual mechanisms.

Another key problem is how to interleave thinking (search in a mental, symbolic problem space) and action (search in an external, physical environment). The two types of processes differ in a variety of ways, most importantly in that a return to a previous state can be achieved by fiat in the internal search space, but has to be accomplished through physical action in the external environment. We intend to experiment with multiple schemes for controlling the interleaving in multiple task domains.

Conclusions

An intelligent agent cannot be limited to learning from positive experience. When task environments change, the extrapolation of prior experience to cover future situations inevitably leads to errors, mistakes and unacceptable outcomes. To exhibit human-level flexibility, an artificial agent needs learning mechanisms that specify how to change in the face of such negative outcomes. The constraint-based specialization mechanism provided this capability in a production system framework before, and we implemented it with the hierarchical skill representation in the ICARUS architecture successfully, after resolving multiple conceptual problems. We performed some test runs in the Blocks World and a navigation domain, and found the mechanism successfully removes failures after revisions. We also verified that the mechanism works well in parallel to another learning mechanism, allowing further study of human level flexibility in this direction.

Acknowledgments

This research was funded by Award # N0001-4-09-1025 from the Office of Naval Research (ONR) to the second author. No endorsement should be inferred.

References

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum.

Ellman, T. (1989). Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, *21*(2), 163–222.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Langley, P. (1987). A general theory of discrimination learning. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development* (pp. 99–161). Cambridge, MA: MIT Press.

Langley, P., & Choi, D. (2006). Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, *7*, 493–518.

Ohlsson, S. (1996). Learning from performance errors. *Psychological Review*, *103*, 241–262.

Ohlsson, S. (2008). Computational models of skill acquisition. In R. Sun (Ed.), *The cambridge handbook of computational psychology* (pp. 359–395). Cambridge, UK: Cambridge University Press.

Ohlsson, S., & Rees, E. (1991). Adaptive search through constraint violations. *Journal of Experimental & Theoretical Artificial Intelligence*, *3*, 33–42.

Wusteman, J. (1992). Explanation-based learning - a survey. *Artificial Intelligence Review*, *6*(3), 243–262.