Cognitive Flexibility through Learning from Constraint Violations

Dongkyu Choi
Stellan Ohlsson
Department of Psychology
University of Illinois at Chicago
1007 W Harrison Street (M/C 285)
Chicago, IL 60607
312-355-0486, 312-996-6643
dongkyuc@uic.edu, stellan@uic.edu

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

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 on the implementation of a constraint-based mechanism for learning from negative outcomes in well-established cognitive architecture, ICARUS. We discuss the challenges encountered during the implementation, describe how we solved them and provide an example of the integrated system's operation.*

1. Background and Rationale

An important goal in the computational modeling of higher cognition is to invent techniques that enable computer programs to mimic the broad human functionality that we call *adaptability*, *flexibility*, or *intelligence*. Cognitive flexibility is a multi-dimensional construct. In this paper, we focus specifically on the ability of humans to act effectively and purposefully even when a familiar task environment is changing, thus rendering previously learned skills and strategies less effective or even obsolete.

When the environment changes, the execution of previously acquired skills is likely to generate actions are that are inappropriate, incorrect or unhelpful vis-à-vis the agent's goal. A key component of flexible adaptation to the changing circumstances is therefore the ability to recover from and unlearn unsuccessful actions in the service of more effective future behavior (Ohlsson, 2010). This problem differs from the standard view of skill acquisition in two principled ways. First, instead of learning a new skill from scratch, the learning agent in this scenario needs to revise an existing skill or strategy. Second, whereas most work in computational modeling of skill acquisition has focused on how to make use of positive outcomes, the adaptation scenario requires mechanisms for learning from errors, mistakes and other types of negative feedback (Ohlsson, 2008).

In past work, we developed a mechanism for learning from negative outcomes that is called *constraint-based specialization* (Ohlsson, 1993, 1996, 2007). This mechanism assumes that the agent has access to declarative knowledge in the form of constraints, where a constraint consists of an ordered pair with a relevance criterion and a satisfaction criterion, <R, S>. Unlike propositions, constraints do not encode truths, but norms and prescriptions, e.g., traffic laws. A speed limit does not describe how fast drivers are going, but specifies the range within which their speeds ought to fall. Constraints support evaluation and judgment rather than deduction or explanation. In a constraint-based system, the architecture matches the relevance criteria of all constraints against the current state of its world in each cycle of operation. For constraints with matching relevance conditions, the satisfaction conditions are matched also. Satisfied constraints require no response, but constraint violations signal a failed expectation (due to a change in the world or to incomplete or erroneous knowledge); this is a learning opportunity. The purpose of the change triggered by a constraint violation is to revise the current skill or strategy in such a way as to avoid violating the same constraint in the future. The computational problem involved in unlearning an error is to specify exactly how to revise the relevant skill when an error is detected. The constraint-based specialization algorithm is a general solution to this problem (Ohlsson & Rees, 1991).

The constraint-based specialization mechanism was previously implemented in HS, a production system architecture (Ohlsson, 1996). The HS system was limited along several dimensions. First, HS did not explicitly represent or take into account the hierarchical

organization of skill knowledge. Second, HS did not explicitly distinguish between search in a mental problem space and search through the environment via overt actions. The implementation of the constraint-based learning mechanism operated with a simple credit/blame assignment rule: Assume that the last production rule to fire before the discovery of an error is the faulty rule. Finally, the HS model only learned from its errors. It is more plausible that human-level flexibility is achieved through the interactions among a set of learning mechanisms, different mechanisms making use of different types of sources of information (Ohlsson, 2008). At the very least, a powerful learning agent should be able to make use of positive as well as negative outcomes.

In this paper, we report preliminary progress in implementing the constraint-based specialization mechanism for learning from error in ICARUS, a cognitive architecture with hierarchical skill knowledge that interleaves thinking and action and that already has a well-developed capability of learning from positive outcomes (Langley & Choi, 2006a). We first describe the relevant features of the ICARUS architecture. We then describe the challenges encountered in implementing constraint-based specialization within ICARUS, with particular attention to the credit assignment problem. Finally, we report an illustrative example of the extended ICARUS, discuss related approaches and outline future work.

2. The ICARUS Architecture

Cognitive architectures aim for a general framework for cognition. An architecture implements as a set of cognitive hypotheses, covering representation, inference, execution, learning and other aspects of cognition. Soar (Laird et al., 1986) and ACT-R (Anderson, 1993) are the most well-known cognitive architectures. ICARUS exhibits some similarities to them, but some differences as well (Langley & Choi, 2006b). Both Soar and ACT-R are rule-based systems, but ICARUS represents skill knowledge differently. Also, ICARUS incorporates a highly developed semantic memory that forces all conceptual knowledge to be grounded in perceptual primitives. In this section, we review the fundamental aspects of the architecture.

2.1 Representation and memories

ICARUS distinguishes conceptual and procedural knowledge. *Concepts* are used to describe the environment around ICARUS, and 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 needs to instantiate them to apply them to a particular situation. Instantiated concepts and skills are short-term structures, in that they are applicable only at a specific moment in time. ICARUS has four separate memories to support these distinctions; see Figure 1.

	Long-term Knowledge	Short-term Structures
Conceptual Contents	Long-term Conceptual Memory	Short-term Conceptual Memory
Procedural Contents	Long-term Skill Memory	Short-term Skill Memory

Figure 1: ICARUS' four-way classification of memory structures.

All concepts are introduced via definitions. Concept definitions are similar to horn clauses, and consist of a head and a body that includes perceptual matching conditions and reference to other concepts. Definitions that do not refer to other concepts define *primitive concepts*. Table 1 shows four ICARUS concept definitions. (Question marks indicate variables.) The first and second concepts have only perceptual matching conditions in their :percepts and :tests fields, so they are primitive. The third and fourth concepts, however, are non-primitive, because they have references to other concepts in their :relations fields. Percepts and tests access ICARUS' environment directly, so their implementation depends on whether ICARUS operates in a simulated or real environment.

Table 1: Sample ICARUS concepts in a Blocks World.

```
((holding ?block)
 :percepts  ((hand ?hand status ?block)
             (block ?block)))

((hand-empty)
 :percepts  ((hand ?hand status ?status))
 :tests     ((eq ?status 'empty)))

((clear ?block)
 :percepts  ((block ?block))
 :relations ((not (on ?other ?block))))

((stackable ?block ?to)
 :percepts  ((block ?block)(block ?to))
 :relations ((clear ?to)(holding ?block)))
```

ICARUS' skills resemble STRIPS operators. The head of each skill is the predicate it is known to achieve. Its body consists of perceptual matching conditions, non-primitive preconditions, and references to either subgoals or direct actions to the world. *Primitive skills* are actions that the agent can execute in the world, whereas non-primitive actions operate on ICARUS' subgoals. The hierarchical organization provides multiple layers of abstraction in the specification of complex procedures. In Table 2, the first skill, which achieves the goal to *hold a block*, has two perceptual preconditions, one of the being that there is a block within reach, one non-primitive precondition, *pickupable*, and two primitive actions, **grasp* and **vertical-move*. The second skill also has perceptual and non-perceptual preconditions but poses two subgoals, *clear* and *holding*, which, in turn, evoke other skills. Because procedures refer to other procedures, the entire set of procedures in long-term skill memory form a hierarchical organization.

Table 2: Sample skills for ICARUS in Blocks World

```
((holding ?block)
 :percepts  ((block ?block)
              (table ?from height ?height))
 :start     ((pickupable ?block ?from))
 :actions   ((*grasp ?block)
             (*vertical-move ?block (+ ?height 50))))

((stackable ?block ?to)
 :percepts ((block ?block)
            (block ?to))
 :start    ((hand-empty))
 :subgoals ((clear ?to)
            (holding ?block)))
```

During performance time, the architecture instantiates these long-term knowledge structures based on the current situation. The bottom-up application of concept definitions creates beliefs in the form of instantiated conceptual predicates and stores them in the short-term conceptual memory (a.k.a. the system's *belief state*). During execution, ICARUS finds executable skills to achieve its goals, and stores the instantiations of those skills in its short-term skill memory. For this reason, procedural short-term memory is sometimes referred to as ICARUS' *goal memory*. In the next section, we explain the system's processes in more detail.

2.2 Inference and execution

The ICARUS architecture operates in cycles. On each cycle, the system creates its current belief state by applying its concept definitions, and decides what to do next by finding a path through its skill hierarchy from its top goal and down to some executable action. When it finds such a path, the system executes the actions proposed by the primitive skill instance that is the leaf node of the path. Figure 2 shows the overall process. The rectangular shapes represent memories in ICARUS with the exception of the one to the far right, which represents the environment. The oval shapes stand for processes that process the information in the memories, while the arrows show the flow of information.

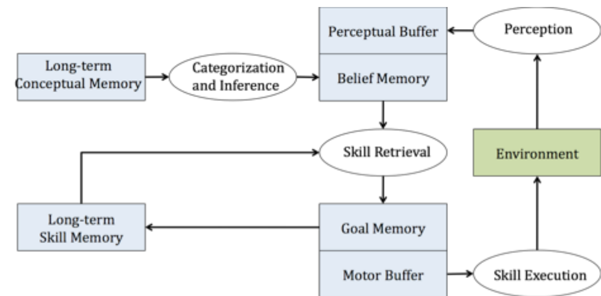

Figure 2: ICARUS' operation in cycles.

The inference process starts with the perceptual buffer, which contains information about the environment. The system attempts to match its concept definitions to the perceptual information. When there is a match, the system instantiates the head of the definitions to compute the current belief state. ICARUS deposits the instantiated concepts in its short-term conceptual memory (a.k.a. its *belief memory*), and it uses those believes during thinking and decision making.

The skill retrieval makes use of several different sources of information. First of all, the process uses the top-level goals specified in the goal memory to guide the retrieval process. It also accesses the contents of the long-term skill memory as well as the current belief state. The system finds relevant long-term skills for its goals, based on the current belief state. Once it finds an executable path through its skill hierarchy from goal to primitive actions, ICARUS performs those actions and thereby changes the environment. The system then starts another cycle, once again beginning by re-computing its current belief state.

3. Learning From Errors

When ICARUS cannot find a skill path from its current goal to an executable action, it invokes a means-ends problem solving capacity that has been described in prior publications (Langley & Choi, 2006a). If it can solve its problem, it captures the solution in the form of new skills that are added to the long-term procedural memory. In this way, ICARUS' stock of skills grows over time.

However, the means-ends based problem solving and learning capability does not enable ICARUS to recover when the environment changes and some of the previously learned skills become incorrect or obsolete.

We extended the ICARUS architecture by incorporating the constraint-based specialization mechanism originally developed for rule-based systems. This required adding a new representation to allow explicit descriptions of constraints and processes that apply constraints to the current belie state. As a consequence, the system can now detect its failures as constraint violations. We then implemented the constraint-based specialization algorithm that allows ICARUS to revise its skills based on its constraint violations.

3.1 Representation of constraints

The 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. ICARUS keeps a list of such pairs in a separate *constraint memory*, which users define in advance. Table 3 shows some examples of constraints that we imposed on the Blocks World domain. The first constraint, *color*, has a single relevance condition, *(on ?a ?b)*, and a satisfaction condition, *(same-color ?a ?b)*. It says that two blocks should have the same color if one is stacked on the other; that is, all the blocks in a tower should the same color. The second constraint, *max-tower*, has a high-level relevance condition and a single satisfaction condition. This constraint restricts the maximum height of towers to three blocks. In constraint language: *A tower should not be higher than three blocks*. Similarly, the third constraint decrees that there should be no other block on top of a particular block designated as a *top-block*, while the fourth says that a block that is stacked on top of another block should be smaller in size than the one it rests on. In constraint language: *Blocks should be stacked in the order of decreasing size*. The predicates used to define the constraints are, like all predicates in ICARUS, defined in terms of other predicates and/or perceptual primitives.

Table 3: Four constraints from the Blocks World.

```
(color :relevance ((on ?a ?b))
        :satisfaction ((same-color ?a ?b)))
(max-tower :relevance ((three-tower ?a ?b ?c ?t))
            :satisfaction ((clear ?a)))
(top-block :relevance ((top-block ?b))
            :satisfaction ((clear ?b)))
(width :relevance ((on ?a ?b))
        :satisfaction ((smaller-than ?a ?b)))
```

3.2 Detection of constraint violations

ICARUS creates its belief state anew on each cycle. It then goes on to retrieve, instantiate and execute one or more skill paths based on the computed beliefs. To learn from errors, the system performs an additional step between inference and execution: It checks if the 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.

We distinguish two different types of constraint violations. In the first type, a constraint becomes relevant but not satisfied. 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. But its satisfaction condition, *(same-color A B)*, is not met in this instance, 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 is relevant and satisfied, but becomes unsatisfied as the result of an action or an environmental event. An example of this type occurs in our constrained Blocks World 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 during 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, so the constraint is violated.

When the architecture finds one or more violated constraints of either type, it invokes the skill revision process to constrain the skill that it just used. The details of the revision process differ between the two types of constraint violations, and we cover both in the following section.

3.3 Skill revisions

Once the system detects constraint violations, it attempts to make revisions to the skill just used. The revision process we use shares its basic steps with those used in previous research with production systems (Ohlsson, 1993, 1996, 2007; Ohlsson & Rees, 1991). The goal of the revision process is to constrain the application of the skills to situations in which it will not violate the constraints. This is done by adding preconditions. The key question is which conditions to add.

The architecture randomly chooses one of the detected violations and attempts to make two revisions by

adding preconditions computed based on the type of the violation. For a type A violation, in which 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 the constraint irrelevant, and another that ensures that the constraint stays satisfied. Table 4 shows how the system computes the new preconditions for the two types of violations.

Table 4: New preconditions created in response to constraint 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, 1993, 1996; Ohlsson & Rees, 1991).

Type \ Revision	1	2
A	not $(C_r - O_a)$	$(C_r - O_a) \cup (C_s - O_a)$
B	not C_r	$C_r \cup$ not $(C_s \cap O_d)$

3.4 Challenges for re-implementation

The differences between the ICARUS architecture and production system architectures force some important changes in the revision process. These pertain to the hierarchical organization of skill knowledge, the definitions of actions and the use of disjunctive definitions.

(a) Hierarchical representation. ICARUS' hierarchical organization of skill knowledge poses one of the most significant changes, in relation to the assignment of credit/blame: which skill should be revised upon detecting a constraint violation? Production systems are flat structures, and it is frequently 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 of the violated constraint, the most reasonable skill to revise might be at the top, at the bottom, or anywhere in between. No simple attribution rule will be sufficient.

In the Blocks World, for example, the system may cause a violation of the *color* constraint by stacking a red block on top of a blue block using the primitive skill, *stacked*. However, the context in which the system executed this particular skill varies based on the situation. Figure 2 shows an example, in which ICARUS did this to achieve its goal, *(color-sorted)*. Here, the last action before the violation occurs is generated by a skill path, *(color-sorted) – (one-color-sorted red) – (on A B) – (stacked A B)*. If the system blindly chose the last skill on this path to revise, it would revise *stacked*. This will not prevent similar violations in subsequent runs, since the system decides which blocks to stack further up in the skill path, namely within the skill, *one-color-sorted*. Therefore, the right skill to revise is *one-color-sorted* rather than the primitive skill, *stacked*. This conclusion is obvious to a human observer in this particular case, but the question is how ICARUS can identify the right skill to revise in the general case.

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. The additional preconditions are fully instantiated at this level, and, therefore, it is the highest level in which all the additional preconditions become meaningful, and it is the right level at which to make the corresponding revisions. For instance, the additional preconditions for the case depicted in Figure 2 are null for the first revision and *(same-color A B)* for the second one. Since a null precondition means no revision, the system makes only one revision in this case. The variable bindings involved in this revision are A and B, and the highest level where both of these are instantiated is at the skill, *(one-color-sorted red)*, which binds its two variables, *?block1* and *?block2* to A and B, respectively. By making a revision at this level, the system checks if the two objects, A and B satisfy the additional condition *(same-color A B)* as soon as they are bound, and prevents the violation of the *color* constraint before it happens. The results of running ICARUS with this solution in place indicate that it is successful.

(b) Add/delete lists. Another problem occurs during the logical 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.

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 but not in the opposite way, making it safe. However, we should compute the *delete* list explicitly due to its use in the revision computations. 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. Similarly, this makes the revisions more restrictive, but not more general, keeping the agent safe, because the delete list is negated during the computation of preconditions.

(c) Disjunctive definitions. ICARUS' support for multiple, disjunctive definitions of concepts adds another layer of complexity. During the operations that compute 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. We changed the architecture to accept 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. But the architecture learns multiple revisions from this case, covering all possible disjunctions of the concept. This approach is based on the understanding that there must be a good reason why such definitions have the same head, thereby creating disjunctions, and that the system benefits from learning about all such cases when, in fact, the current situation involves only one of them. 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. In everyday language, we would refer to this as understanding the situation.

4. An Illustrative Example

In this section, we provide an example that illustrates the operation of the extended ICARUS system. We use the Blocks World that has served as an initial test bed during our development and implementation of the system. It supports many constraints with various complexities, and yet it stays relatively simple and easily understandable. We created four different constraints for this world as shown in Table 3.

The system has a skill set that is general in the sense that the skills do not have special preconditions that ensure the satisfaction of the constraints. For example, the system would know how to stack a block on top of another, but does not know if the skill would or would not cause any violations of *color*, *max-tower*, *top-block*, or *width* constraints. We gave the system opportunities to experience several different initial conditions and goals that naturally lead to violations of these constraints, and ICARUS learned revisions based on the violations. The experience eventually resulted in a successful run until completion of its top-level goals.

Fig. 3 shows some sample runs where the architecture achieves its goal, *(color-sorted)* in three runs. During the first run, the system stacks a blue block D on top of a red block B. The width of block D is smaller than that of block B, but the colors of them are different, violating the *color* constraint,

(on D B) → *(same-color D B)*

From this error, the architecture learns a revised version of its non-primitive skill, *one-color-sorted*, with an additional precondition, *same-color*. During the second run, the system incurs yet another error and violates the *width* constraint,

(on E D) → *(smaller-than E D)*

Then the system revises another skill with the same head, *one-color-sorted*, to include an additional precondition, *smaller-than*. After that, the system may or may not experience further failures that involve other constraints, but eventually it succeeds in achieving its top-level goal, as shown in the third run. We reset the initial tabletop state between runs, enabling the system to restart from the initial conditions without the need to undo what it has done so far. The puzzle-like characteristics of the Blocks World make this reasonable.

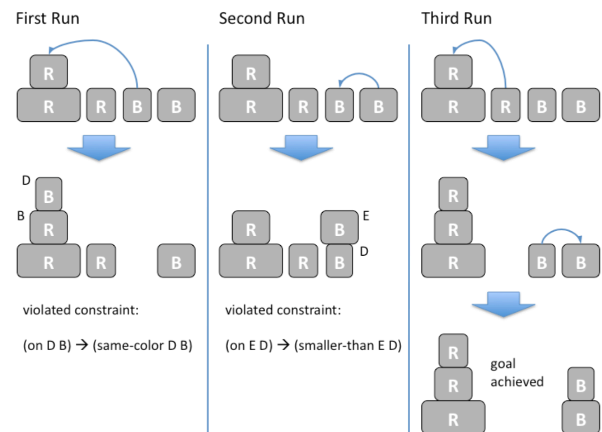

Fig. 3. Two learning events that lead to a successful run in the Blocks World.

In short, the solutions to the challenges posed by the architectural characteristics of ICARUS appear to be successful. The hierarchical organization of skill knowledge forces the question of at which level the revisions are to be applied. The principle that they apply at the level at which all relevant variables are bound has so far selected the right level in all simulation runs. Comparing successive belief states

appears to serve instead of explicit *add* and *delete* lists. Finally, ICARUS' support for multiple, disjunctive definitions of concepts poses the problem of which disjunctions to include in a revision. Our solution to a include all of them brings with it a modes form of understanding, because it allows the system to know what to do in situations it has never seen before.

5. Related Work

Two types of error correcting mechanisms have been developed in prior work, weakening and discrimination. The idea behind weakening is that when a knowledge structure (rule, skill element, schema, chunk, etc.) contributes to the production of an action, which, in turn, generates a negative outcome, then the strength associated with that knowledge structure is decreased according to some function. Weakening is not a powerful mechanism, because actions are not typically correct or incorrect, or appropriate or inappropriate, in themselves. Instead, actions are appropriate, correct or useful in some situations but not others. The goal of learning from error 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 any type of situation.

In the 1980s, Langley (1987) proposed a computational model of discrimination. The key idea behind this contribution was to compare situations with negative and positive outcomes to identify discriminating features. The SAGE system stored every application of every production rule in memory. If an action generated both positive and negative outcomes across multiple situations, the situation features that were true for one type of outcome but not for the other were identified and used to constrain the applicability of the rule. The problems with this computational discrimination mechanism include (a) 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; (b) 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 (c) the inability to identify potential discriminating features with a causal impact from those of accidental correlation.

The production system implementation of constraint-based specialization overcame most of these weaknesses. Unlike weakening, it identifies the specific class of situations in which an action is likely (or unlikely) to cause errors. Unlike Langley-style discrimination, constraint-based specialization does not carry out an uncertain, inductive inference, but computes a rationally motivated revision to the current skill. These advantages were limited by a simplistic credit/blame attribution algorithm and a lack of learning mechanisms for capitalizing on successful outcome. The implementation of constraint-based specialization within the ICARUS architecture has removed those limitations.

6. Future Work

A key problem is to study the interactions among multiple learning mechanisms. People learn in a variety of ways (Ohlsson, 2008) and human-level flexibility is the outcome of the interactions among the multiple mechanisms. Our current understanding of how learning mechanisms interact to produce flexible behavior is limited. We intend to add additional learning mechanisms to ICARUS, including mechanisms for learning from examples and from analogies, and explore the conditions under which multiple mechanisms produce more flexible behavior than single mechanisms. A second key problem is how effectively to interleave thinking – i.e., search in a mental, symbolic problem space – and action – i.e., 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.

7. Conclusion

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, a computational agent needs learning mechanisms that specify how to change in the face of such negative outcomes. The constraint-based specialization mechanism has been shown to be successful when implemented in a production system architecture. Its implementation with the hierarchical skill representation in the ICARUS architecture posed multiple conceptual problems. The most important of these was the assignment of credit/blame in a hierarchical system. That is, how to locate the right level in a skill path at which to apply the new constraints? The answer is that the constraints apply at the level at which all the relevant variables were bound. Some test runs in the Blocks World support this idea. This solution has the advantages of being easily computable and general across domains. The possibility that it applies to other types of hierarchical systems might deserve attention.

8. References

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

Anderson, J. R. (2007). *How can the human mind occur in the physical universe?* New York: Oxford University Press.

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. (1983). Learning search strategies through discrimination. *International Journal of Man-Machine Studies, 18*, 513-541.

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, *9*, 217-260.

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

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

Langley, P., & Choi, D. (2006b). A unified cognitive architecture for physical agents. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 1469-1474.

Ohlsson, S. (1993) The interaction between knowledge and practice in the acquisition of cognitive skills. In A. Meyrowitz and S. Chipman (Eds.), *Foundations of knowledge acquisition: Cognitive models of complex learning* (pp. 147-208). Norwell, MA: Kluwer.

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

Ohlsson, S. (2007). The effects of order: A constraint-based explanation. In F. E. Ritter, J. Nerb, E. Lehtinen and T. M. O'Shea (Eds.), *In order to learn: How the sequence of topics influences learning* (pp. 151-165). New York, NY: Oxford University Press.

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. (2010). *Deep learning: How the mind overrides experience*. New York: Cambridge University Press.

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

Acknowledgement

The work reported in this paper was supported by Award # N0001-4-09-1025 from the Office of Naval Research (ONR) to the second author. No endorsement should be inferred.

Author Biographies

DONGKYU CHOI is a Visiting Research Specialist in the Department of Psychology at the University of Illinois at Chicago (UIC). He is in the process of completing his Ph.D. degree from the Department of Aeronautics and Astronautics at Stanford University, specializing in cognitive architectures. His work at UIC includes postdoctoral research with Stellan Ohlsson.

STELLAN OHLSSON is Professor of Psychology and Adjunct Professor of Computer Science at the University of Illinois at Chicago (UIC). He has held academic positions at the University of Uppsala, Sweden, Carnegie-Mellon University, and the Learning Research and Development Center at the University of Pittsburgh. He has published widely on issues pertaining to skill acquisition, creative insight, the design of intelligent tutoring systems and other cognitive research topics.