Modeling Instruction Fetch in Procedural Learning

Bryan Stearns (stearns@umich.edu)
John Laird (laird@umich.edu)
University of Michigan, 2260 Hayward Street
Ann Arbor, MI 48109-2121 USA

Abstract

Cognitive architecture agents execute and compile known procedures to model how humans learn procedural skill knowledge for a given task. It is often assumed that agents have fully learned how to order and condition that execution before the task begins. Is this assumption valid, or can it impair human modeling efforts? We posit that the first of the classic three phases of skills necessitates learning how to order task execution and that models should account for this process. We evaluate the effects of modeling this process using a general fetch and execute learning agent in Soar and apply it to modeling two different human studies. Our agent introduces a procedural chunking method for autonomously learning declarative memory structures by which spreading activation can guide task-specific execution order. We demonstrate that modeling the process of learning how to order task execution can significantly improve human model results.

Keywords: skill acquisition; control; phases; spreading activation; cognitive architecture; Soar; primitive elements; PROP.

Introduction

Procedural skill knowledge is considered a fundamental component of human cognition. How the human mind learns and organizes such knowledge has been of scientific interest for many years. Cognitive architectures, such as Soar and ACT-R, provide a means to improve our understanding of the mind through computational simulation of the processes thought to underly cognition. Some consensus has arisen in the community regarding certain qualities of cognition that these systems both reflect (Laird, Lebiere, & Rosenbloom, 2017).

One theory of skill learning that has been operationalized in ACT-R is the classic three phases proposed by Fitts and Posner (1967). In the *cognitive phase*, the mind learns how to understand task execution by deliberately reasoning over each cue, action, and desired outcome. Task responses might be incorrect while the correct ordering of actions is learned. The *associative phase* follows once practical task understanding is achieved. Responses to task cues become increasingly fluid with practice. In the final *autonomous phase*, those responses are automatic and occur by reflex, subject to little cognitive control or interference, such that unrelated reasoning can occur in parallel.

In ACT-R, *cognitive phase* learning is modeled via procedure compilation (Tenison & Anderson, 2016). Procedures are represented as rules, and during practice new rules are created by combining or specializing existing rules for more efficient task operation. It models *associative phase* learning by strengthening declarative memories, such that they are more readily available after practiced access. An ACT-R agent reaches the *autonomous phase* when it has learned task-specific rules that incorporate declarative knowledge, such that declarative retrievals are not needed.

Primitive elements theory (Taatgen, 2013) describes how architecturally primitive procedural knowledge can be compiled into useful, transferable skills through practice. This learning can be applied to any task using a general *fetch and execute* cycle. Knowledge of how to practice a skill is fetched from long-term declarative memory, and the indicated skill is then practiced and learned. When primitive elements theory was implemented in the Actransfer architecture, a variant of ACT-R, it produced good models of both human learning and transfer in many specific domains (Taatgen, 2013).

Primitive elements theory has also been applied to the Soar cognitive architecture, leading to the Soar PRimitive OPerator (PROP) model of primitive learning and transfer (Stearns, Assanie, & Laird, 2017). The PROP model includes an additional process of generating working memory addresses as a primitive skill necessary for instantiating general procedural knowledge. This inclusion improved power-law learning behavior to better match human data.

However, all these methods assume that *when* to fetch skills is learned before model behavior begins. This assumption is common in cognitive modeling, and underlies designs that hard-code task conditions into initial procedural knowledge. When to fetch and how to execute are learned separately, and fetching is abstracted away from models.

In this paper we consider whether that assumption neglects an important aspect of *cognitive phase* learning: learning how to arrange task execution correctly. How might models that learn skill fetching alongside execution differ from those that ignore fetching? Does this assumption impair human modeling? To shed light on these questions, we extend the original PROP model to learn to fetch *during* skill execution. We show that modeling how skills are fetched during task execution can significantly improve simulations of human behavior. For simplicity, we will refer to the original PROP model as PROP_1, and our model that learns to fetch as PROP_2.

Fetch and Execute

PROP_2 is defined for Soar, in which procedural knowledge is encoded as *if-then* rules, and describes how a task-specific skill can be converted from declarative into procedural knowledge through repeated practice of a task-general *fetch and execute* cycle.

PROP_2 progresses through the three phases of skill learning. In the *cognitive phase*, the agent knows rules for the process of fetching and executing instructions, but does not know rules for when specific instructions should be fetched and applied. Until it learns to fetch, a process we describe later, fetching is random and can retrieve non-applicable in-

structions. After the agent gains enough experience to always fetch applicable instructions, it is in the *associative phase*. In this phase, it learns increasingly task-specific rules that reduce instruction execution latency. Eventually the agent learns rules to perform a task automatically without fetching, and is then in the *autonomous phase*.

We assume that instructions are stored in long-term declarative memory by some prior learning process. Interactive task learning (Laird, Gluck, et al., 2017) is one means by which an instructor can teach instructions to an agent.

Instructions include three components: conditions, actions, and any literal values used within them. Conditions and actions describe primitive assembly-like operations supported by the architecture, and these are uniquely identified by the working memory elements they use. For example, instructions might load the string "hello" (a literal value) into memory element A, test equality of values in elements A and B (a condition), and then copy the value from B to C (an action).

After fetching, the agent executes an instruction by first evaluating the instruction conditions. Initially this is done with innate primitive rules. One rule fires to test the first condition, another tests the second condition, and so on. If all are true, additional rules execute the actions. If any are false, the agent returns to fetching to attempt different instructions.

When the agent executes a sequence of instructed operations, it automatically learns rules that combine those operations together, as pictured in Figure 1. This process drives the bulk of learning during the *associative phase*.[1] Sequentially associated rules are combined into new rules, which fire to execute instructions in fewer steps than required by primitive rules. These rules are general and can transfer to execute any instructions that invoke the same architectural operations, but their structure is specific to experienced task operations.

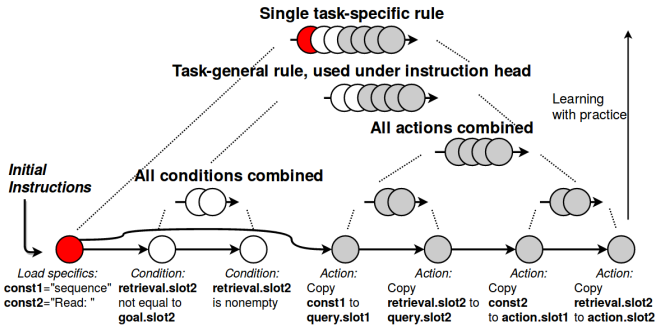


Figure 1: Hierarchical clustering of primitive memory operations, adapted from (Taatgen, 2013). The architecture iteratively combines seven primitive task-general operations through practice into a single task-specific rule. In this example, actions query for the next item in a sequence, while printing the current value.

Taatgen (2013) showed that hierarchical learning of primitive memory operations provides a good model for human skill learning and transfer in many domains. For good transfer, it is important that rules are built *gradually* over iterations of execution. A new rule only takes effect after multiple learning attempts so that only rule combinations that appear often across instructions are used within the skill hierarchy.

[1] In agreement with ACT-R, our model also considers declarative strengthening to be *associative phase* learning.

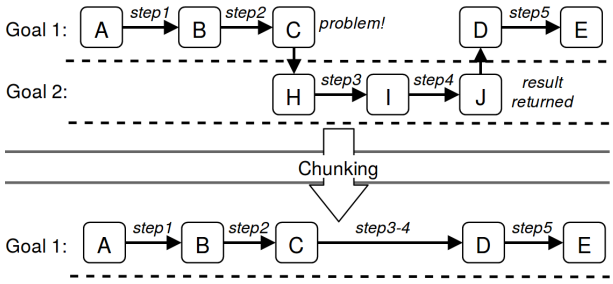


Figure 2: Soar chunking. Goal 2 is created as a subgoal of Goal 1. Goal 2 decision making eventually derives a result that allows Goal 1 to continue. Chunking learns this result as a rule that recreates the solution (step3-4) the next time the scenario arises.

Associative rule composition eventually builds a single task-specific rule equivalent to the complete fetched instruction set. Once learned, this rule will execute the instructed conditions and actions by reflex without the need for reasoning over fetched instructions. An agent that primarily performs a task by using such rules is in the *autonomous phase*.

This process defines how task rules are learned, and ignores how the agent knows what to fetch and execute at a given moment. But, as we will show, the same methods can be used for learning what to fetch. In the next section, we describe gradual procedural learning in Soar, the mechanism we use for hierarchical associative learning and for learning to fetch.

Gradual Procedural Learning in Soar

ACT-R learns procedural knowledge by a hierarchical compilation process similar to Figure 1. In contrast, Soar learns procedural knowledge by summarizing problem solving.

Soar manages a stack of agent goals and corresponding working memory partitions. Newer subgoals in the stack represent subproblems of earlier goals. As shown in Figure 2, whenever agent reasoning connects declarative knowledge from a subgoal's memory to a higher goal's memory ("returns a result"), the architecture summarizes subgoal decision making by creating a new rule that recreates that result whenever the situation that led to the subgoal arises again. This learning is called *chunking* in Soar, and learned skills are called *chunks*.[2] Chunking is normally one-shot, and once a result is returned, the corresponding chunk will preempt future subgoal problem solving. To implement our model, we extended Soar to include gradual chunking.

To chunk gradually, the architecture tracks the number of times a specific chunk is submitted for creation and only adds that chunk to procedural memory if that number passes a parameterized threshold.[3] Setting the threshold to 1 is equivalent to standard one-shot chunking. A threshold of 2 would

[2] Not to be confused with ACT-R declarative *chunks*.

[3] Other methods can easily be used in place of creation counts, but this is sufficient for our purposes here.

require a subgoal to be solved twice, and so on.

To build the skill hierarchy shown in Figure 1, the $PROP_2$ model includes innate rules that explicitly combine pairs of invoked operations together as results for chunking after they are executed. These combined operations can then be used to apply instructions in future iterations of execution.

Learning to Fetch in Soar

An agent might have many instructions in long-term declarative memory that it could fetch at any time. When it needs to execute and learn a new skill, how does it know what to retrieve? Instructions will not be applicable unless their described conditions are true, but how can the agent know if the conditions for specific instructions are true until after it fetches and evaluates them? Without a method for biasing memory retrievals toward instructions that are likely to have matching conditions, all of declarative memory might have to be searched before usable instructions are found. While the order of operations can be hard-coded into rules, we argue that learning what to retrieve is an important aspect of *cognitive phase* learning.

The Actransfer architecture controls fetching by precalculating which conditions were true for each instruction that might be retrieved, and boosting the activation for instructions with satisfied conditions (Taatgen, 2013). The agent then fetches instructions with the highest activation value.

$PROP_1$ implemented instruction fetching by deliberately controlling the fetch sequence. Declarative knowledge of the correct instruction sequence for a given task (e.g. "$Skill1 \rightarrow Skill45 \rightarrow Skill2 \rightarrow Done$") was provided to an agent at the same time as those instructions. The agent then iteratively fetched each element of that sequence to perform a task. This method required the agent to constantly track its current position in that sequence. A problem with that approach is that the agent loses track of its position in the sequence if it enters the *autonomous phase*. The task-specific rules that drive behavior in that phase act by reflex outside fetching and the agent lacks meta-knowledge of what rules it fires. Thus, the agent will not know to update its sequence position. When it does need to fetch, it will resume where it last left off in the declarative sequence, and will sequentially fetch and unsuccessfully evaluate all operations that were just performed autonomously until it catches up to the actual state of the task. Deliberate fetching becomes a performance bottleneck by requiring cognitive control over selecting each task step, even when that step has already been performed.

Neither of these approaches explains how skill fetching is learned. Rather than assuming activation boosts as in Actransfer or relying on a fixed fetching sequence as in $PROP_1$, we create a more comprehensive model that learns how to fetch instructions. We implement this by using Soar chunking to learn connections for spreading activation.

Primitive elements theory proposed using spreading activation to guide instruction fetching, though this was not implemented in Actransfer (Taatgen, 2013). By this method, when a query to long-term declarative memory has multiple possible results, the memory with the highest activation is retrieved. Spreading activation increases the activations of memories associated with the current working memory context. Working memory elements connected to long-term memory elements boost the activations of those memories according to connection weights. If those long-term memories are connected to other long-term memories, that boost spreads, with some decay, to also increase their activations, and so on to other memories.

Inspired by primitive elements theory, and the recent addition of spreading activation in Soar (Jones, Wandzel, & Laird, 2016), we developed a novel model for learning to fetch. This model follows three constraints of Soar theory: First, only working memories that also exist in long-term memory can be sources of spreading activation. Second, Soar models a fan-effect by normalizing a memory's spread over the number of its descendants in the long-term memory graph. Third, such working memories are either created through long-term memory retrievals or through chunks.[4]

To learn spreading, $PROP_2$ learns to create or remove long-term memory elements within working memory, causing spread according to the first constraint. Each spread source corresponds to an individual condition described by an instruction, and spreads to that instruction. The agent learns to create a spread source whenever the corresponding condition is true and remove it when it is false. Thus, instructions with the most true conditions are favored for retrieval.

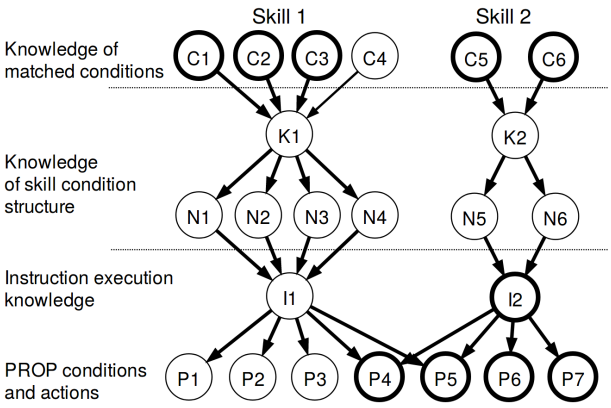


Figure 3: Declarative structures for skill fetching in Soar. Nodes in bold are present in working memory. Spread from these is normalized over the number of a rule's conditions. Since condition C4 is false, I1 receives 3/4 spread while I2 receives 2/2. I2 is then fetched.

But the agent should ideally retrieve instructions with *fully* satisfied conditions. If all else is equal, instructions with only six of seven conditions met should receive less total activation than those with two of two conditions met. Therefore, according to the second constraint, we structure long-term memory so that spread normalizes over the number of conditions in a set of instructions, as shown in Figure 3. In the figure, three

[4]Creating such memories through chunks is a feature of Soar 9.6, developed by Mazin Assanie.

of four conditions are known to be satisfied for Skill1, and two of two conditions for Skill2. The execution knowledge for Skill1 thus receives 3/4 normalized spread while that for Skill2 receives 2/2 spread. Skill2 is therefore fetched.

To satisfy the third constraint, we use chunking to learn *when* to create knowledge of satisfied conditions. When evaluating the conditions from fetched instructions, if the agent finds a true condition, it returns declarative knowledge of that result. After instructions have been fetched and evaluated enough times over the course of a task to satisfy the chunking threshold, Soar automatically chunks a rule that recreates this result whenever the condition is met. As soon as that condition is no longer met, the chunk no longer matches, and the spread source is removed from working memory.

For simplicity, we provide long-term declarative knowledge of conditions at the same time as instructions, structured in the format shown in Figure 3. Our evaluation is concerned with learning to create knowledge of matched conditions within working memory.

Our agent begins each fetch with an open query for instructions that can be biased by spreading. Results will be random at first before conditions are learned. If our agent finds that it retrieved non-applicable instructions, it reverts to cognitive control for fetching by retrieving and following the explicit declarative fetch sequence. As the agent learns conditions through experience, spread provides sufficient bias such that controlled fetching becomes unnecessary.

Parameters for Skill Fetching

We have described our task-general design for learning skill fetching and execution. Our aim is to evaluate the importance of modeling the fetch process. We define two boolean parameters for modeling fetching, which in the $PROP_2$ model determine what kinds of rules an agent learns.

The first parameter is either **LEARNED** or **KNOWN**, and determines whether an agent learns fetch order during task execution. If fetching is LEARNED, the $PROP_2$ agent will learn rules for spreading during execution. If fetching is KNOWN, the agent will *always* fetch through cognitive control by following a declaratively known fetch sequence.

The second parameter is either **AUTO** or **DELIBERATE**. If set to AUTO, the agent eventually learns a single rule that will perform task operations by reflex (the top-most composition depicted in Figure 1), bypassing declarative instructions and cognitive control. If set to DELIBERATE, this final composition step is prevented, so that execution must *always* be deliberate and fetching cannot be bypassed.

Evaluation

We evaluate these parameters against human behavior using two human domains: a text-editors task Singley and Anderson (1985) and a mental arithmetic task (Elio, 1986). The editors task examines transfer, while the arithmetic task focuses on the learning curve. These have already been modeled in Actransfer, allowing us to compare our $PROP_2$ model results with those from the alternate fetch/execute paradigm. Actransfer is a KNOWN-AUTO model, since it assumes fetching order (by hard-coding activation behavior) and allows task rules to be fully learned, so ideally our model should be similar to Actransfer when using KNOWN-AUTO.

We implement the same Actransfer agent designs, using supplementary materials from (Taatgen, 2013), including the same declarative instructions and the same simulated timing for memory retrievals and vision/motor latencies. We also model time using the same 50 msec per decision as is common in Soar and ACT-R.[5]

To select the gradual chunking threshold, we performed a threshold sweep (not shown) to find the value that provides the closest fit to the Actransfer model for comparison. Matching differences in Actransfer model learning rates, we use a threshold of 48 for editors task models, and 10 for arithmetic task models. We also match Actransfer by averaging performance over 12 trials for the editors task and 8 trials for the arithmetic task. These models are fairly deterministic, and variance is low.

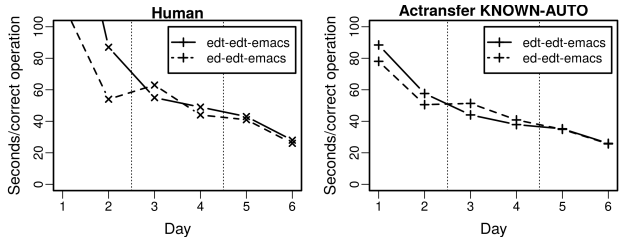
We compare our results with average human performance rather than that of individuals. Taking the average abstracts away any individual differences among humans that would arise from lifelong learning experiences that preceded participation in the studies. Modeling average human performance is our first step, and modeling individual differences is beyond the scope of the current study, although we plan to study it in the future.

Editors Task

In the editors task, typists modified documents according to written edit directions. Example directions are to replace one word with another or to delete a sentence. Three keyboard-only editors were used with which participants had no prior experience: ED, EDT, and EMACS. These use different keyboard commands, and ED and EDT also differ from EMACS by being simpler single-line editors. The experiment took place over six days, with some participants switching editors after two days to test transfer. If a participant spent two days each on ED, EDT, and EMACS in that order, we call this case ED-EDT-EMACS. If initial performance in EDT was faster after using ED than when using EDT on day one, this indicated transfer to EDT. We focus on transfer of ED to EDT-EMACS, but other editor permutations show similar results.

Figure 4a shows results from the human experiment. Performance on EDT after two days of ED is almost as good as after two days of EDT, indicating substantial transfer. There is similarly significant transfer to EMACS on day five. (EMACS users required about 80 sec on day 1, not shown). Figure 4b shows the Actransfer model. Model performance is fast during days 1-2, but transfer trends are the same. Figure 5 shows the parameterized $PROP_2$ models. Transfer is similar among all models in Figures 4 and 5.

[5]Soar uses decisions to carry out retrievals, and we replace the 50 msec from those decisions with retrieval time.



(a) Human performance (b) Actransfer model

Figure 4: Human data from (Singley & Anderson, 1985) and the Actransfer model from (Taatgen, 2013), demonstrating transfer between editors.

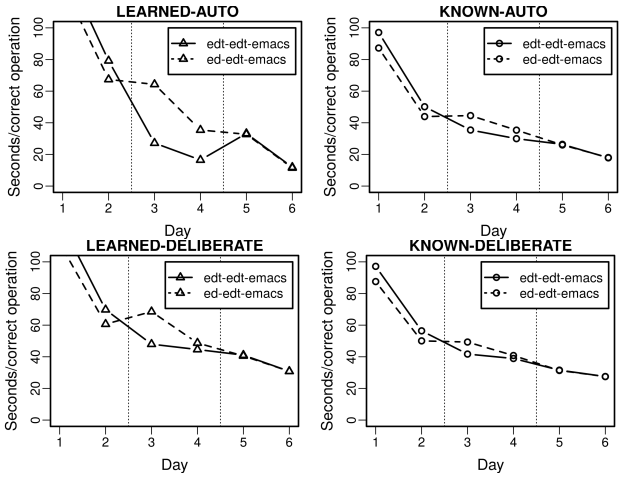


Figure 5: $PROP_2$ models of editors, varying over LEARNED (left column), KNOWN (right column), AUTO (top row), and DELIBERATE (bottom row).

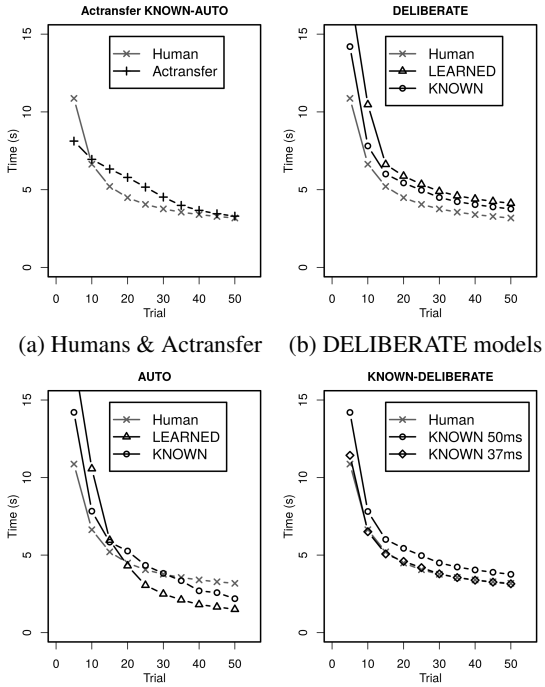
In Figure 5, observe that there is a significant difference between LEARNED (left column) and KNOWN (right column) results, primarily in slower LEARNED performance on days 1-3. These are cases when non-controlled fetching retrieves mostly random instructions. Interestingly, after fetch learning, LEARNED models are not far behind the KNOWN models in performance, demonstrating that LEARNED models learn fetching and execution simultaneously.

Also notice that AUTO models (top row) achieve super-human performance by day 6, at under 20 sec. Human and Actransfer models, by contrast, end near 30 sec, as do our DELIBERATE models (bottom row). The Actransfer KNOWN-AUTO model uses precalculated activation to fetch, and does not exhibit this phenomenon. But we surmise that Actransfer is slower than our AUTO on day 6 due to activation noise, a part of that model that can also cause incorrect fetching.

Overall, we see that LEARNED-DELIBERATE (bottom left) most closely matches humans by accounting for slower performance during the first days (due to LEARNED), while finishing at the correct performance on the last day (due to DELIBERATE). Performance on days 1-3 is too fast for EDT but slightly slow for ED, implying the instructions for EDT should be more complex and those for ED slightly less so.

Arithmetic Task

In the arithmetic task, human subjects memorized a mental arithmetic algorithm and applied it to provided inputs for 50 trials. For brevity we omit transfer details and focus on the learning curve.



(a) Humans & Actransfer (b) DELIBERATE models

(c) AUTO models (d) Best human model

Figure 6: Models of the (Elio, 1986) arithmetic experiment.

Figure 6a shows results for humans and the Actransfer model. Only the power-law fit to human performance was available from the original study, and this is what is shown as the human model.

Figure 6b shows our LEARNED- and KNOWN-DELIBERATE models. We first note the power-law performance of $PROP_2$, which is from having to learn how to access working memory for tasks, inherited from $PROP_1$ (Stearns et al., 2017). We also see clearly how the LEARNED model catches up to the KNOWN model after 15 trials. It does not catch up entirely, due to the fuzzy nature of using activation.

Figure 6c shows our LEARNED-AUTO and KNOWN-AUTO models. As with the editors task, the AUTO parameter results in eventual super-human performance. The KNOWN-AUTO learning curve is not smooth, due to times when autonomous task-specific rules invalidate the controlled fetch sequence, which must then be stepped through to catch up to the task state. But we observe that after trial 20 the LEARNED-AUTO agent acquires enough fetch experience to not require further deliberate fetch control.

We also observed that while model time incorporates many factors, such as retrievals and motor latency, the amount by

which KNOWN-DELIBERATE behavior differs from human performance is a multiple of decision cycle time. Setting decision cycle time to 37 msec results in almost an exact fit to the human curve, as shown in Figure 6d.[6] While not a standard timing, this fits neural modeling that predicts cycle times of approximately 40 msec for simple actions (Stewart, Choo, & Eliasmith, 2010). Table 1 shows the mean-squared-errors of various timing models compared with the human model.

	Actransfer	**50 msec**	**40 msec**	**37 msec**
MSE	1.270	1.695	0.177	**0.0382**

Table 1: Mean Squared Errors for Actransfer and KNOWN-DELIBERATE models with varying simulated times for decisions.

Discussion

The above parameters reflect Fitts and Posner's (1967) three stages of skill learning in our model. The *cognitive phase* is represented when fetch LEARNING is enabled. Hierarchical compilation of instruction execution demonstrates the *associative phase*, and allowing AUTO task-rules to be compiled leads to the *autonomous phase*.

We model *cognitive phase* learning with a novel use of Soar chunking by which chunks create or retract sources of spreading activation. This general learning method could be applied whenever there is an occasion for learning context-appropriate retrievals, not just fetch modeling.

In the editors task, LEARNED fetching accounts for initial human behavior in ways KNOWN fetching could not. However, in the arithmetic task, KNOWN models were most accurate. We believe this reflects the natures of the tasks. Human subjects performed the arithmetic task *after* being trained in the algorithms, while editors subjects memorized only an editor's individual keyboard commands prior to experimentation. Subjects performing the arithmetic experiment should therefore have already mostly completed the *cognitive phase*, which is not the case for subjects in the editors experiment.

Though AUTO allows faster execution, we notice that DELIBERATE achieves the closest human performance in both experiments. Within our model this implies that humans do not perform these tasks entirely by reflex after training, but continue to reason over each step. This might imply that the *autonomous phase* is more appropriate for modeling motor skills, as Fitts and Posner were evaluating, rather than dominantly cognitive skills.

Our arithmetic task model is almost identical to the human model when decision cycle time is just under 40 msec. By contrast, changing cycle times has little effect on the editors model, since it performs at the scale of 100 sec, largely due to memory retrieval times. Editors agents also employ a higher chunking threshold than used by arithmetic agents, implying that human processing is more complex for the editors task than for the arithmetic task compared to our models, which also makes sense given the different time scales. The appropriate complexity of task models and the validity of 40 msec cycles for primitive skills are beyond the scope of this paper. However, they present intriguing avenues of study.

We therefore conclude that consideration for the stages of learning and these fetching parameters is critical for human modeling. Whether an agent learns fetching or assumes it to be already learned should reflect the task being modeled. The fetch process of choosing what to execute is one by which the human mind controls its own behavior. Learning that control can be important for even simple modeling, as we have demonstrated, but understanding the theory of that learning could play an important role in unraveling the mysteries of human cognition.

[6]This would not be achieved by changing the chunking threshold, as that alters the learning curve shape in addition to scale.

Acknowledgments

The work described here was supported in part by the Office of Naval Research under Grant Number N00014-18-1-2010. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the ONR or the U.S. Government.

References

Elio, R. (1986). Representation of similar well-learned cognitive procedures. *Cognitive Science*, *10*(1), 41 - 73.

Fitts, P., & Posner, M. (1967). *Human performance*. Belmont, CA: Brooks/Cole Pub. Co.

Jones, S. J. M., Wandzel, A. R., & Laird, J. E. (2016). Efficient computation of spreading activation using lazy evaluation. In *International conference on cognitive modeling.*

Laird, J. E., Gluck, K., Anderson, J., Forbus, K. D., Jenkins, O. C., Lebiere, C., ... Kirk, J. R. (2017). Interactive task learning. *IEEE Intelligent Systems*, *32*(4), 6-21.

Laird, J. E., Lebiere, C., & Rosenbloom, P. S. (2017). A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine*, *38*(4), 13-26.

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies*, *22*(4), 403 - 423.

Stearns, B., Assanie, M., & Laird, J. E. (2017). Applying primitive elements theory for procedural transfer in soar. In *International conference on cognitive modeling.*

Stewart, T. C., Choo, X., & Eliasmith, C. (2010). Dynamic behaviour of a spiking model of action selection in the basal ganglia. In *International conference on cognitive modeling* (pp. 235–40).

Taatgen, N. A. (2013). The nature and transfer of cognitive skills. *Psychological Review*, *120*(3), 439–471.

Tenison, C., & Anderson, J. R. (2016). Modeling the distinct phases of skill acquisition. *Journal of experimental psychology. Learning, memory, and cognition*, *42 5*, 749-67.