A Comprehensive Computational Model of PRIMs Theory for Task-Independent Procedural Learning

by

Bryan Stearns

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor John E. Laird, Chair
Professor Nikola Banovic
Emeritus Professor David Kieras
Professor Priti Shah
Professor Niels Taatgen, University of Groningen

"Stimulus-response theory and production systems represent an approach to procedural knowledge which has not been very popular in cognitive psychology. They attempt to analyze all procedural knowledge into a fixed set of primitives. These primitives are the atoms out of which a great variety of cognitive behavior is constructed – language comprehension, language generation, inference making, question answering, problem solving, executing instructions, etc. If such a set of primitives can be found, this would seem to constitute an enormous advance because the few principles that describe these primitives could be used to account for all of cognitive behavior."

(John R. Anderson, 1976, p. 80)

Bryan Stearns
stearns@umich.edu
ORCID iD: 0000-0002-2422-9286

Acknowledgments

I must first thank my Creator and Lord, Jesus Christ, for teaching me and lovingly providing for me throughout this whole journey.
I also must thank my family profusely for their endless love, support, and patience; my father, who showed me perseverance with a sense of humor, my mother, who showed me gentleness and critical thinking, and my sister, who showed me joy and the constant spark of creativity. I love you all very much.
I would especially like to thank my advisor and mentor, John, for his tireless coaching and counsel and support. It is an understatement to say I could not have done this without him. And I am very grateful to my committee, Nikola, David, Priti, and Niels, for their patience and feedback and counsel, and for the hours they have contributed to the completion of this work. I must also thank my compatriots in the Soar lab, especially Steven, Peter, and Mazin, for their friendship, feedback, brainstorming sessions, and for showing me the ropes. You guys are the best.
And a big thank you must go to my bride, Rachel, who has supported and encouraged me especially through these last months. You have made this journey worthwhile.

The work described here was supported in part by the ONR under Grant Numbers N00014-15-1-2058, N00014-18-1-2010, F048875-093099, and the AFOSR under Grant Number F050045-094301. 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 ONR, AFOSR, or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

MAE Mean Absolute Error

MAPE Mean Absolute Percentage Error

PFC Pre-Frontal Cortex

PRIM Primitive information processing element

PROP Primitive Operator

PSCM Problem-Space Computational Model

LTDM Long-term Declarative Memory

RITL Rapid Instructed Task Learning

RL Reinforcement Learning

SMEM Semantic Memory

WM Working Memory

ABSTRACT

The human ability to reason about and learn practically any task has been studied for countless years, but to date we still do not truly understand how human learning is task-independent at the computational level. Researchers have theorized that we can account for many human cognitive behaviors if we combine a task-independent set of primitive procedures with a robust, general learning mechanism that compiles them into cognitive skills for various tasks. The PRIMs theory of procedure learning and transfer is a cognitive architecture theory of human learning that shows how a task-independent set of primitive procedures can support learning in any task that is also supported by the underlying architecture. However, its published architecture implementation, Actransfer, focuses on modeling transfer and does not specify all of the computational details of PRIMs theory.

This thesis presents a computationally comprehensive cognitive architecture model of PRIMs theory that I call the PROPs system. I comprehensively define each of the processing steps that PRIMs theory requires and implement these in an agent model using the Soar cognitive architecture. I do this through a methodology for incrementally refining a cognitive architecture model. I use this methodology to extend PRIMs theory and unify it with three-phase learning theory from human performance research, task set theory from psychology and neuroscience, and Soar theory from cognitive architecture research. This achieves several improvements in the model's ability to replicate human learning behavior.

Among the contributions of this work, I introduce a novel form of primitive processing that explains the origins of the primitive procedures of PRIMs theory and supports procedural learning in an unbounded, dynamic working memory space. I show that this improves the model's ability to match human power-law learning. I also extend Soar cognitive architecture theory with grad-

ual procedural learning in a manner consistent with Soar's existing theory and introduce a novel computational approach by which a cognitive architecture model can learn to guide automatic long-term declarative memory retrievals based on working memory contents. I finally introduce a novel computational approach by which a model can guide deliberate retrievals through choice-based decision making.

In my evaluation of the PROPs system, I identify ways in which PRIMs theory for procedural learning might be further unified with neuroscience theory to broaden the model to include declarative learning. I also identify boundaries where PRIMs models can or cannot currently account for types of human cognitive processing when the models are constrained to be fully task-independent and consistent with the surrounding cognitive architecture. This reveals a path for future cognitive architecture research and development.

CHAPTER 1

Introduction

Humans demonstrate the epitome of general skill learning. We can enter seemingly any unfamiliar domain and learn how to perform effectively in it in a short amount of time. If you were given the task of transcribing text from a piece of paper into a computer text editor, you could almost certainly do it, even though you were not born with most of the necessary skills, such as how to read the paper and the computer screen, how to use the particular text editor's interface for moving the cursor and editing the document, or how to type accurately and quickly. But while we see the human ability to learn such tasks everywhere around us, we still do not have a detailed understanding of *how* we do it. Whatever the answer, it is clear that humans must have some forms of innate, *task-independent* skills and learning abilities.

This thesis pursues a greater understanding of human task-independent procedural learning. "Task-independent" here refers to the ability to be effective in many tasks without constraint or specialization for any particular task. Rather, the breadth of tasks is unspecified. Task independence is distinct from the ability to function in *every* task domain, which might be referred to as task generality. I pursue task-independence comparable to that shown by humans. "Procedural learning" here refers to the learning of tacit, procedural skill knowledge, such as the skill to effectively type on a computer keyboard, in contrast with explicit, declarative knowledge about task skills, such as the knowledge of how one intends to type on a keyboard. The question of task-independent procedural learning is the question of how humans are able to develop procedural skills for practically any task.

The last half-century of research has seen much progress in the quest for understanding task-independent human learning. Research in procedural learning has theorized that a fixed set of innate, primitive procedural knowledge can be used as the building blocks from which humans learn the great variety of cognitive behaviors that we learn (Anderson, 1987). But while many researchers in the community have done a great deal to forward our understanding of human procedural learning over the course of many decades (Anderson, 1982; Fitts, 1954; Kim et al., 2013), current models still do not provide computationally comprehensive and precise accounts of the primitive procedures that allow humans to function across a wide variety of tasks and of the ways

that humans compose these when learning task skills. The current state-of-the-art in terms of task-independence is the *PRIMitive Elements* (PRIMs) theory put forward by Taatgen (2013). This theory describes how the process of composing cognitive skills from task-independent procedural primitives provides an explanation for human transfer behavior. The original publication of PRIMs theory included a computational cognitive architecture implementation of its principles, called Actransfer, that replicated human learning and transfer trends in many tasks. Unlike prior work, this model defines primitive procedures that can be composed into skills for any task supported by the underlying cognitive architecture. However, its implementation focused more on demonstrating the mechanics of transfer and less on the computation that supports the whole of PRIMs theory, and it left several gaps in its computational explanation of the cognitive processing steps required in the theory.

In this thesis, I extend the work of PRIMs by developing a novel, computationally precise model of task-independent human procedural learning called the *PRimitive OPerators* (PROPs) system. PROPs extends PRIMs theory and its Actransfer model by defining and implementing the gaps in its computational explanation. I extend PRIMs theory by drawing from the computational theory of the Soar cognitive architecture (Laird, 2012), three-phase theory from human skill learning research (Fitts & Posner, 1967; Gray & Lindstedt, 2017), task set theory from psychology and neuroscience (Sakai, 2008), and from other related research into human cognition.

1.1 Model Desiderata

The research presented here comes primarily from the domain of cognitive architectures (Kotseruba & Tsotsos, 2018; Newell, 1990). Cognitive architectures pursue a computational understanding of the structure and processing of general human cognition. Allen Newell, one of the founders of cognitive architecture research, outlined three paradigms for this kind of work in his famous "20 Questions" paper (1973). I constrain the PROPs system to specifically satisfy Newell's first and third paradigms:[1]

> "The first suggestion is to construct complete processing models rather than the partial ones we now do. [...] The attempts in some of the other papers to move toward a process model by giving a flow diagram (Cooper-Shepard and Klahr) seem to me not to be tight enough. Too much is left unspecified and unconstrained. To make the comparison with Chase and Simon somewhat sharp, these flow diagrams are not sufficient to perform their tasks."

[1] Newell's second paradigm concerns the coordination of psychological experiments alongside modeling and is beyond the scope of this thesis.

> "The third alternative paradigm we have in mind is to stay with the diverse collection of small experimental tasks, as now, but to construct a single system to perform them all. This single system (this model of the human information processor) would have to take the instructions for each, as well as carry out the task. For it must truly be a single system in order to provide the integration that we seek."

I extract four desiderata from these paradigms:

D1. **The model must be comprehensive.** That is, it must specify and constrain the details of any cognitive processing that is necessary for its execution.

D2. **The model must be able to perform its tasks.** That is, it must be possible to implement it within an input/output environment to test its learning behavior in practice.

D3. **The model must be task-independent.** That is, it must be implemented as a single system that is used to perform all experimental tasks, with variation only in the form of different task instructions given to the system.

D4. **The model must be a model of human information processing.** That is, the model's computation must be consistent with the current understanding of the human cognitive architecture.

The first two desiderata derive from Newell's first paradigm, in which he champions "complete processing models." First is that the model be *comprehensive*, such that it enacts all the computation that is described in its theory. Second is that the model be able to perform in an input/output environment. Note that while Newell did promote the pursuit of a complete end-to-end model of all processes needed between environment input and output, D2 only constrains this thesis work to be embedded within a system that can interact with the environment so that its cognitive processing can be applied in simulated experiments. A complete model of human perception or motor learning, for instance, is outside the scope of this thesis. The third desideratum reflects Newell's third paradigm, in which he calls for a *general single system*, which can carry out a variety of tasks after receiving the different instructions for each. The variation between task models should only be in terms of the task instructions given to the computer program. The process by which those instructions are interpreted and carried out should be common across tasks. The fourth and final desideratum flows from Newell's third paradigm as well as the overall mission of the field, which is to make a single computational system that models the *human information processor*. While it might be the case that there are other computational approaches that could achieve human-like learning other than the approach used by humans (whatever that approach might be), there are no other known demonstrations of human cognitive capabilities in nature. Our lack of understanding

in how all the components of cognition fit together constrains me to be as faithful as possible to the human standard as I pursue an understanding of human procedural learning.

To satisfy D1, it is not enough to merely supply a *functional* model that generates PRIMs theory's predicted behavior if it does not implement the details of the theorized processing. I explain this idea by an example. Assume that you wish to model a theory of human decision making for the task of transcribing text that was mentioned in the first paragraph of this thesis, and one decision you wish to model in this task is the choice between either copy-pasting or retyping text. You theorize that when subjects make this choice, they first imagine the futures of each potential action and how long each would take, and then they choose the one with the least expected effort. Assume your theory also predicts that the time required for decision making is a function of the number of steps the person requires to imagine each outcome. You might develop a computer model of this theory in which the computer agent "imagines" how long each edit operation takes by retrieving the answer from a provided lookup table. Then you might simulate the latency of the agent's decision making based on a mathematical function of the lookup table values that the agent examined. This kind of model implementation might provide the outward functional behavior that your theory predicts. However, it does not perform the computation that your theory prescribes because the model does not imagine the futures of its actions to generate its decision, and its timing is not a direct function of the agent's processing steps. You might be tempted to build your model this way because the computational details of imagination are too complex or unknown to implement, but omitting these from the implementation means that the model computation does not completely simulate the *processing* prescribed by your theory. For my model to be computationally comprehensive, I must implement and demonstrate the processes that are prescribed by the theory I am modeling.

D2 and D3 are relatively straightforward today given the significant prior work that has been done in this domain. To satisfy D2, I embed my model within the Soar cognitive architecture, which provides an interface for models to interact with environment input and output, and I use PRIMs theory to define a task-independent processing by which to apply Soar's mechanisms for human-like learning and transfer. The main contributions of this thesis stem from combining these with D1. I develop the PROPs system to represent a comprehensive computational model of the task-independent primitive procedural learning outlined by PRIMs theory and embedded within the interactive processing of Soar.

Where D1 constrains the model to implement computation for each step of PRIMs theory, D4 constrains the model so that its implementation is consistent with what is known about actual human processing and the human architecture. My model of procedural learning would not be conducive to understanding human cognition if its computational details were inconsistent with human processing. For instance, the current consensus of the Common Model of Cognition (Laird,

Lebiere, et al., 2017; West, 2020) is that human procedural memory systems do not directly interact with motor systems but rather that any interaction must first pass through working memory and requires deliberate decision making. I use decades of research in human processing and behavior to inform my model's computational implementation, both from the human theory integrated within the Soar cognitive architecture and the Common Model and from other disciplines of human research.

In short, my goal is to develop a comprehensive computational model of all the internal processing steps described by the PRIMs theory of procedural learning and transfer. In order to function as a model of learning, it must use these steps to actually perform real tasks, and it must do so using computation that represents a consistent model of task-independent human cognition.

1.2 Research Approach

As stated above, I develop my model within the Soar cognitive architecture, and primarily draw from cognitive architecture theory to constrain and define the details of the PROPs system. I select the Soar cognitive architecture for my work because of its focus on computational science and its applicability for real artificial intelligence while also modeling human intelligence.

As I describe in more detail in chapter 3, I use an iterative approach to develop a comprehensive implementation of PRIMs theory. I begin by replicating the original Actransfer implementation in Soar without attempting to add computational detail beyond what is necessary for the system to perform the tasks. Wherever Actransfer used computation that produced necessary functional behavior but which did not reflect the processing steps prescribed by PRIMs theory, I do the same. I then incrementally fill or replace the incomplete portions of the implementation with novel, detailed computation, and evaluate each change relative to the performance of the prior iteration.

Soar as an architecture is task-independent, but to make a system that satisfies D3, the agent design that runs using the architecture must also be task-independent. While this should follow naturally from implementing the overall agent design of PRIMs theory, I take care to preserve task-independence in my Soar agent design through each iteration of development.

The PROPs system does not attempt to address learning that is outside the scope of PRIMs theory, such as declarative learning that constructs explicit knowledge about what to perform in a task. Further, in order to prevent assumptions about knowledge representation from interfering with the model of task-independent primitive skill composition, I constrain PROPs to use innate, architectural processes and innate knowledge alone to model task learning. That is, I do not program any PROPs models with procedural knowledge if in theory that knowledge must at some point be learned. If it is necessary to use more complex, composed knowledge to model a task, to represent knowledge that a task subject would already know before beginning the task, then I

am constrained to first compose that knowledge through agent learning rather than through hand-written assumption. I can then copy the knowledge that the agent learns and use it as assumed starting knowledge for a particular task if it makes sense for that task's starting conditions or if I wish to disable the process of learning that knowledge during evaluation. Thus, even if I provide the system with non-innate knowledge for a task, it will derive from the same model of primitive procedural learning that I develop in this thesis.

1.3 Evaluation Approach

The original Actransfer implementation of PRIMs theory was evaluated with a suite of four human learning and transfer experiments involving six different tasks (Taatgen, 2013). This demonstrated both the various aspects of PRIMs theory and its task independence. The same system performed each task with the only major variation being the task-specific instructions and environments; however, some architecture parameters were varied for each task as is not uncommon in cognitive modeling.

To ensure that the PROPs system satisfies D3 and maintains the same task-independence and generality prescribed by PRIMs theory, I apply PROPs to the same suite of experiments, similarly varying only the task instructions and environments across tasks. This provides evidence that PROPs has the same capabilities. In order to isolate and evaluate the effects of my implementation changes, to the furthest extent possible, I use not only the same suite of experiments as Actransfer but also the same task instructions and environments. Thus, the computation that implements the instructions is the only variable when comparing the different models.

Before I can evaluate PROPs with respect to D1, specifically how comprehensively it implements the steps of PRIMs theory, I must first formally define the processing steps prescribed by PRIMs theory. I do this in chapter 4. I then determine to what extent Actransfer implements all of these steps. For each iteration of developing the PROPs system, I then determine the extent to which it also implements these steps. We consider an implementation of PRIMs theory complete if it implements all steps according to this specification. The formal specification of Actransfer and PROPs' implementation of the PRIMs flow diagram supplies evidence that PROPs sufficiently implements each step to satisfy D1. The experimental comparison with Actransfer for each iteration then reveals any behavioral or theoretical consequences of the differences.

As I explore in section 2.3, this sort of internal processing comparison has, to my knowledge, rarely, if at all, been attempted before in cognitive modeling research and demonstrates an evaluation approach that could be more broadly used in the research community to compare competing theories and their implementations.

Finally, as I evaluate the effects of each implementation change, I compare PROPs system

performance with human performance in these same tasks. Though I use Actransfer as a reference for testing the effects of computational modifications, my goal is that these modifications result in equivalent or superior fits to human performance.

1.4 Contributions

There are two categories of contributions from this work. The first category is for the set of theoretical advances introduced by the PROPs model that result from addressing the challenges I faced in defining the details of its computation. I present five main theoretical contributions in this category. These are the main contributions of this thesis. The second category is for the contributions that can aid future work in cognitive modeling, as described in section 1.4.2.

1.4.1 Theory Contributions

The PROPs system represents a theory of computation for general procedural learning that extends PRIMs theory in four chief ways. I summarize these here along with the section reference where each is explained in the thesis:

- **I introduce a new layer of primitive procedural memory processing to the production system architecture model of procedural learning.** I demonstrate that this additional layer results in a more human-like power-law learning and transfer profile that is absent from Actransfer results. (7.1)
- **I introduce a new approach by which a cognitive model learns to use the state of working memory to guide long-term declarative memory retrievals.** I demonstrate that this learning process replicates aspects of human behavior that previous models could not. (8.1.2)
- **I extend the Soar cognitive architecture to support gradual procedural learning in a manner consistent with existing architecture theory.** I demonstrate that this can replicate the gradualness of human learning. (8.2)
- **I introduce a novel cognitive architecture model of human task sets.** I demonstrate how this can model human choice-based decision making, task switching, and working memory interference effects. (9.1.1-9.1.2)

There are other minor extensions or novel distinctions that I introduce to PRIMs theory that I discuss in this thesis. my evaluations focus on the four contributions listed above, however.

1.4.2 Modeling Contributions

The work of developing the PROPs system results in the following two secondary contributions, which can assist with future cognitive modeling research:

1. I develop the PROPs system code as a robust, task-independent tool for developing cognitive model agents.[2]

2. I define and demonstrate a two-part methodology for developing and evaluating incremental advances to cognitive architecture theories. (chapter 3)

First is the PROPs agent code, which future researchers can use for general cognitive modeling in Soar. While Soar has been used for cognitive modeling practically since its inception, PROPs provides a structured and constrained task-independent framework for applying Soar to human behavior modeling. Agent programmers can leverage the PROPs system's built-in processing for procedural learning, transfer, and decision making, and need only encode task-specific instructions using a high-level problem space description. This simplifies agent development considerably compared to most cognitive architecture agent implementations, which require one to define all system-level behavior for the architecture mechanism interactions. The PROPs system provides these out of the box in a manner that is consistent with human cognitive theory. In computer science terms, this is analogous to providing a higher-level programming language like Python compared to a more fundamental language like C.

Second is the methodology I use for developing and evaluating PROPs as an implementation of PRIMs theory. As described in the next chapter, there is little precedent in the literature for how to compare my work with prior cognitive architecture research. This contribution comes in two interrelated parts. First is a methodology for evaluating my architecture model against a prior model. Second is a methodology for extending that prior model. These are defined and integrated in Tables 3.1 and 3.2 on page 19.

[2]This is publicly available at https://github.com/Bryan-Stearns/PROPs.

1.5 Outline

The rest of this dissertation is organized into the following sections:

2. **Background.** I summarize related work in computational models of task-independent human skill learning.

3. **Methodology.** I describe my approach for developing the PROPs system and for evaluating it in terms of my desiderata.

4. **PRIMs and Actransfer.** I give an overview of PRIMs theory and a specification of its processing steps that I implement with PROPs. I also discuss how PRIMs has been implemented in Actransfer and the ways I need to expand upon this implementation.

5. **Actransfer Experimentation.** I describe the Actransfer experiment suite, which I use to evaluate PROPs, as well as Actransfer's original performance.

6. **Soar.** I describe the relevant theoretical and computational principles of the Soar cognitive architecture to the extent that they inform my design of the PROPs system.

7. **PROPs Iteration 1: Defining Support for Working Memory Operations.** I describe the first iteration of the PROPs system, which replicated Actransfer processing in Soar and introduced a new layer of primitive processing for that purpose.

8. **PROPs Iteration 2: Defining Declarative Retrievals.** I describe the second iteration of the PROPs system, which introduced gradual procedural learning in Soar and extended PRIMs theory and computation for memory retrievals using related work in cognitive science.

9. **PROPs Iteration 3: Defining Decision Making and Timing.** I describe the third and final iteration of the PROPs system, which unified Soar theory with PRIMs theory, integrated task set theory for human decision making and task switching, and completed the computational model that I present here in this thesis.

10. **Discussion and Related Work.** I discuss the PROPs system in connection to related research and review outstanding questions and areas of potential future work.

11. **Conclusion.** I briefly summarize this work's contributions and directions for further study.

CHAPTER 2

Background

In this chapter, I review three lines of research relevant to this thesis. First is the field of cognitive architecture. Second is research in general human skill learning and the associated cognitive processes. Third is the work related to the problem of evaluating models of cognition, which has been a difficult endeavor since the beginnings of cognitive modeling.

2.1 Cognitive Architecture

To paraphrase Laird, Lebiere, et al. (2017), a cognitive architecture defines a general purpose computational device capable of running programs on data, with the difference being that the programs and data are limited to those appropriate for human-like intelligent behavior and are ultimately intended to be learned from experience rather than programmed, aside from possibly a limited set of innate programs. Commonly, the programs they run for different tasks are referred to as *agents* or simply *models*, and researchers often test different theories of cognition by imbuing agents with knowledge that leads them to behave in the theorized way.

While the field was pioneered by a small handful of architectures such as Soar and ACT-R (Anderson, 2007), today it is estimated that there are close to 300 architectures in existence, with at least a third of them actively used for cognitive modeling (Kotseruba & Tsotsos, 2018). The Common Model of Cognition is an emerging consensus in the cognitive architecture community regarding the general computational framework of human cognitive processing (West, 2020). Figure 2.1 depicts the high-level structure of the Common Model. The two cognitive architectures I focus on in this thesis, Soar and Actransfer (as a derivative of ACT-R), both follow this model. Both use symbolic graph-based data structures to model the contents of Working Memory (WM). Both use *if-then* production rules to represent procedural knowledge and the computational means by which an agent modifies its WM. Example production rules might be, `IF (prompt is shown) THEN (read the prompt),` or `IF (prompt asks for report) THEN (set goal to check progress).` Both architectures also define a

Long-term Declarative Memory (LTDM) where their agents can store or retrieve explicit, symbolic knowledge, such as "The prompt says 'Hello.'" or "$2 + 5 = 7$."

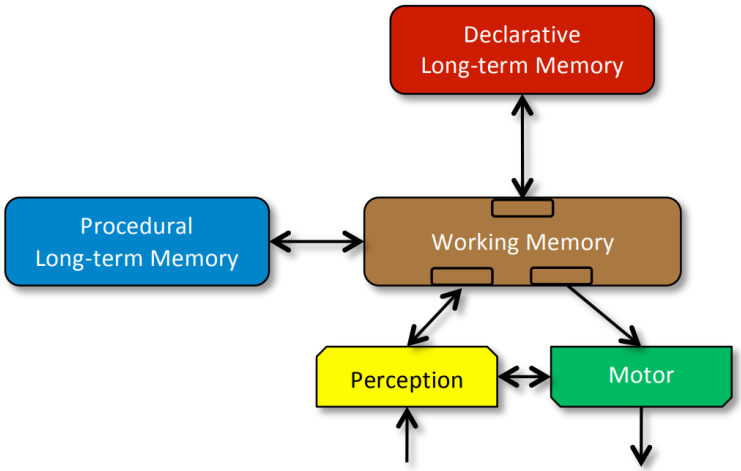


Figure 2.1: The Common Model of Cognition, taken from (Laird, Lebiere, et al., 2017).

To date, cognitive architectures have been used to successfully model a wide variety of human task behaviors. ACT-R is the most commonly used in cognitive modeling, having been used in over 1,100 publications (Ritter et al., 2019). However, most cognitive architecture publications use different agent model designs within the architecture, not a "single system" design that can be applied to model multiple, diverse tasks. But progress is beginning to be made in this direction. Researchers using ACT-R have begun to make use of a common set of agent procedural rules for learning in multiple tasks (Anderson et al., 2019). Similarly, the Soar agent Rosie (Kirk, 2019) is a single system that can learn 60 different games and a variety of mobile robot tasks by instruction, though Rosie has focused on achieving task performance rather than on modeling human learning.

In contrast, work with PRIMs theory (Taatgen, 2013) has provided a cognitive model of procedural learning and transfer that is truly task independent and consistent with the Common Model of Cognition. Agents based on PRIMs theory generate all their behavior from a small set of innate primitive knowledge, and they require only task-specific instructions in order to be applied to tasks supported by the architecture.

Taatgen (2013) embedded PRIMs theory in ACT-R, and the resulting architecture is called Actransfer. Though most of its core processing is the same as that of ACT-R, Actransfer is its own architecture. Actransfer thus generally satisfies D2 as a system that can be applied to real tasks just as ACT-R is, and it also satisfies D3 as a task-independent system that can model a variety of tasks. Further, it is based on a large body of ACT-R and independent human research that implies it is consistent with D4 (Huijser et al., 2018; Taatgen, 2019). However, based on the analysis presented below, Actransfer is incomplete in terms of D1 as a *comprehensive model* of PRIMs theory. Actransfer provides architectural computational for procedure learning and transfer as

prescribed by PRIMs theory, but it elides important details in the theorized processes that support the main theory, particularly in areas of memory use and decision making. As I discuss in chapter 4, it does not entirely satisfy D4 to the extent that it uses ACT-R components in ways that appear to deviate from their intended theory for modeling human cognition. For example, to implement agent decision making, Actransfer employs the ACT-R declarative retrieval mechanism, although in ACT-R theory, a different utility-based process is hypothesized to represent human decision making (Anderson, 2007). It should be noted, however, that sometimes an architecture simply does not provide all the mechanisms that the theory calls for, and to evaluate the theory at all with an existing architecture requires using the existing architecture's mechanisms in a manner contrary to their original design. When this happens, it can motivate a reformulation of the theory that one wishes to model, a modification to the architecture, or else a completely new architecture that implements the theory through a novel approach to representing human cognitive processing. This must be kept in mind with my work with PROPs as well as I apply PRIMs theory in the Soar cognitive architecture.

2.2 Human Skill Learning

A theory of human learning that has had great impact and is still in use today is the three-phase theory of skill acquisition by Fitts and Posner (1967). Three-phase theory correlates outward transitions in learning performance with theoretical transitions in mental processing. In the first *cognitive phase*, a subject is still learning what approach to use when performing the task and must therefore deliberately reason over each cue, action, and desired outcome while performing the task. Thus, this phase is characterized by the need for intense cognitive effort as well as highly variable or incorrect actions while the subject uses trial and error to learn. The second *associative phase* follows once the subject has learned a stable approach for the task and is then learning to perform that approach well. In this phase, repeated practice lets the subject become increasingly fluid and accurate as they gain automatic procedural skills for responding to task cues. In the final *autonomous phase*, the subject has proceduralized their approach enough so that they can perform in the task with expertise almost entirely automatically, subject to little cognitive effort or control or cognitive interference, such that the subject can engage in unrelated reasoning while performing the task. Three-phase theory has influenced most of the other works I describe below. In the years since its introduction, Posner and others have found that each phase corresponds to processing within particular brain regions (Perez et al., 2018). These different brain regions can operate in parallel, such that there is some overlap when progressing from one phase to the next, though as a whole a learner follows an ordered progression through each phase.

Psychology research has also come to distinguish between *declarative* and *procedural* long-

term memory knowledge over many decades of research, and these terms are now well established (Kump et al., 2015; Radvansky & Tamplin, 2012; Squire, 1986, 2004). Generally speaking, declarative knowledge is explicit, factual knowledge about the world. It usually can be verbalized (is declarable) and manipulable by deliberate reasoning. Procedural knowledge, on the other hand, is tacit knowledge for performing cognitive or motor actions, and it can only be accessed by using it for performance. Procedural knowledge is generally skill that is learned through repeated practice, and, as non-declarative memory, it cannot be directly inspected or modified by deliberate reasoning. It is thus resistant to change and lasts a long time compared to declarative memory (Radvansky & Tamplin, 2012). Since the 1980s, it has been understood that procedural knowledge is in fact one of multiple different kinds of non-declarative memory. Other kinds include memory for perception or memory for emotional responses (Squire, 2004). Procedural memory is the memory for skills and habits, both motor skills and cognitive skills.

Anderson (1982) incorporated the declarative/procedural distinction and three-phase theory within his theory for the ACT cognitive architecture, a precursor to ACT-R. ACT represents declarative knowledge with symbolic graph structures, and procedural knowledge with production rules. In ACT, the three phases are called the declarative, transition, and procedural phases. In the first declarative phase, the architecture reasons over and arranges declarative facts about the task domain. In the second transition phase, the agent learns to encode declarative knowledge about procedures into the production rules. Once the agent skill knowledge is primarily represented in production rule form, the learner is said to be in the third procedural phase. This work has been continued and expanded upon by others in cognitive modeling, such as in KRK theory (Kim et al., 2013). KRK theory additionally incorporates a theory of forgetting during the learning phases. The alignment of three-phase theory with stages of declarative and procedural knowledge representation is depicted in Figure 2.2.

The smooth curve in this figure typifies the result of averaging human data across many individuals. A single individual's performance might not be so smooth. The Plateaus, Dips, and Leaps (PDL) framework by Gray and Lindstedt (2017) complements the three-phase theory by describing the nuanced role of progression through the three phases in an individual. In this framework, a learner's progression through the three phases consists of multiple smaller cycles of progression through the phases, marked by plateaus, dips, and leaps in task performance. As a learner gains associative/autonomous phase expertise in a particular task approach, their performance reaches a *plateau*. The subject can then recognize that they must alter their approach to achieve even greater performance, and then re-enter the cognitive phase by attempting a alternate, novel approach. This can lead to a slight *dip* in performance. Then, as the subject gains mastery in their new, superior approach, their performance *leaps* to new levels of mastery. A single subject can exhibit multiple transitions through plateaus, dips, and leaps in a single session of practice.

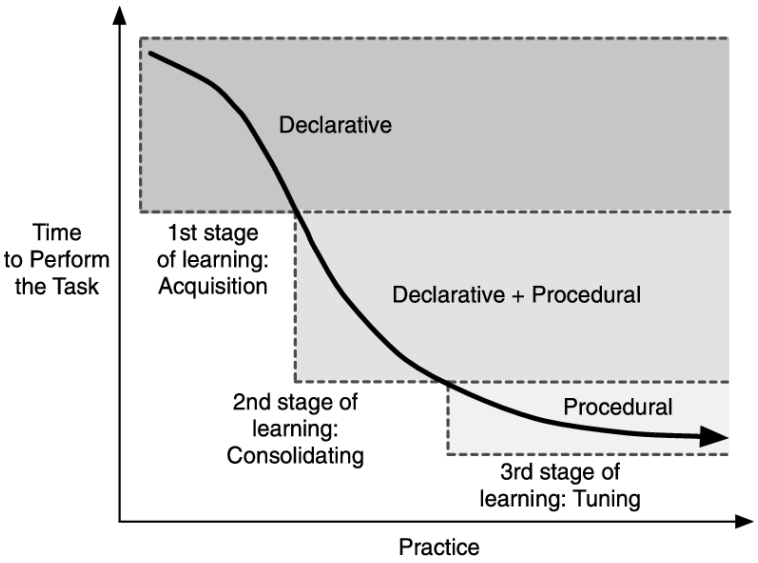


Figure 2.2: How performance changes with practice in three phases of knowledge representation. Figure taken from (Kim & Ritter, 2015), originally used for depicting KRK theory.

Bovair and Kieras (1991) described a model of procedural learning that distinguishes between procedure comprehension and procedure interpretation in the context of learning skills through reading text descriptions of the task. Procedure comprehension is the stage when declarative reasoning constructs an internal mental representation for the approach described in the written procedures, represented in terms of known skill knowledge that can apply that approach. Procedure interpretation invokes these constructions to actually execute the procedures. The procedure comprehension process was specifically described in terms of the cognitive / declarative phase of skill learning.

A significant portion of research in procedural learning concerns the nature of transfer. Broadly speaking, transfer is an improvement in one's ability to learn one skill due to prior learning in another skill. Thorndike (1922) proposed the *identical elements theory* of transfer, which states that a person transfers knowledge from one skill to another because the mental procedures in both invoke the same skill knowledge elements. Singley and Anderson (1985) proposed a computational definition for procedural transfer through the *identical productions* model. This model uses production rules as the identical skill elements that are reused across tasks. When an agent for this model learns a rule for one task that can be reused later when performing a second task, the agent does not need to learn that rule again for the second task, which reduces learning time. Singley and Anderson (1987) evaluated the identical productions model using ACT. By comparing the model with human performance, they found that in some cases the model produced a fairly accurate relative prediction of human data. In other cases, it achieved only half the transfer measured in human participants, indicating that the model was not a complete explanation of transfer.

Bovair and Kieras (1986) analyzed cognitive phase learning in humans and showed that using individual conditions and actions within instructed task rules as identical elements of transfer predicts human transfer fairly closely. Though their work primarily concerned declarative comprehension of task rules, this result supported the notion that individual, generalizable conditions and actions might be identical elements of transfer in human *procedural* learning as well, rather than whole task-specific productions. This follows from the three-phase understanding that humans learn procedural representations from declarative representations. Bovair and Kieras used regression to show how the various factors of task training, such as training order and rule elements, contributed to learning and transfer. For example, subjects for this research would take about 8 sec longer to learn completely novel task rules compared to rules with transferred conditions or actions.

Taatgen (2013) proposed PRIMs theory, which I describe in more detail in chapter 4, as a modification of the identical productions model of transfer that expands the idea of transfer across rule conditions and actions. PRIMs theory describes a task-independent cognitive architecture approach that generates human procedural learning and transfer behavior based on the actual computation of composing task rules from individual condition and action memory operations. These memory operations are task independent because they are defined in terms of the architecture's memory systems (e.g. the general action of copying a value from one particular memory slot to another). These condition and action procedural primitives ("PRIMs") form the basis of an assembly-like language by which any task can be instructed, executed, and learned. Taatgen (2013) used the Actransfer implementation of PRIMs theory to generate human transfer behaviors in a wide variety of tasks.

Like KRK and other theories, PRIMs describes skill learning as a process of learning automatic procedural knowledge from practice, based on declarative knowledge of a task approach. The PRIMs agent begins with declarative instructions for which primitive procedures to execute, and, with practice, the agent's architecture compiles these primitives together into larger, more specialized procedures that provide faster performance. PRIMs, as implemented in Actransfer, achieves more transfer than prior models due to the generality of its primitive production rules.

In its implementation of PRIMs theory, the Actransfer architecture focused on the computational representation of primitive skills through production rules and on how they can be compiled for transfer with practice. However, its implementation elides the computation of some supporting processes, such as the process of how the initial declarative instructions are retrieved into WM for the agent to examine and execute.

A newer evolution of Actransfer has been developed, simply referred to as the PRIMs architecture (Huijser et al., 2018), which has some experimental variations from Actransfer (N. Taatgen, personal communication, Dec 17, 2019). However, publications that use this variation have not de-

scribed its mechanics or evaluated its modifications compared to Actransfer. For this thesis, I focus on the original publication of Actransfer, which more directly reflects the mechanics of ACT-R.[1]

PRIMs theory assumes that a subject has already derived an internal declarative representation for a task approach that they wish to practice, and it only attempts to model the process of compiling that representation into procedural knowledge. As I discuss in section 8.3, this corresponds to the associative/autonomous phases of three phase theory. My work excludes the cognitive / declarative phase and the question of how humans use reasoning or creativity to derive ideas about a task approach. I am concerned primarily with creating a task-independent model of what Bovair calls the procedure interpretation process. This thesis presents a comprehensive computational model of PRIMs theory for procedural learning and transfer during the procedure interpretation process, not a complete model of the entire process of learning task procedures, which would also include what Bovair calls the procedure comprehension process.

2.3 Theory Evaluation

A computational model allows a theory to be implemented and tested in a controlled settings. But one can implement a single theory with different computational models, and not all implementations will necessarily make the same predictions or be consistent with what is known about human processing. Details matter, and the computational details refine and constrain the theory to the extent that they are meant to represent processing done by humans. I must therefore evaluate the merits of my implementation choices.

Researchers often compare different model implementations in terms of how well they reproduce external human behavior for specific tasks. If one model reproduces human behavior more accurately than another, this gives weight to the theory behind that implementation. If both models can generate the same external behavior even when using different computation, it is difficult to evaluate whether either implementation reflects human cognition more accurately than the other, or whether one will be more fruitful for future research. This is an instance of what are known as *identifiability problems* in cognitive modeling (Beck & Chang, 2007).

There are two types of identifiability problems, the *uniqueness problem*, in which many different computational approaches can produce equivalent external behavior, and the *discovery problem*, in which the space of possible solutions is vast but there is little guidance for finding the correct one (Anderson, 1993). Indeed, evaluating models of the internal processes of cognition has been called a form of black-box testing (Ritter et al., 2019), analogous to trying to determine a computer's machine language when the only available data are program outputs (Newell, 1973).

[1] Actransfer was a modification of ACT-R 6.

There has been some prior work comparing cognitive architectures. Wharton and Lewis (1990) compared Soar and the Construction-Integration Model on a trivial button-press task, but the architectural mechanisms were apparently not comparable enough to associate with outward behavioral differences. Johnson (1997) reviewed and compared theoretical assumptions in ACT-R and Soar, but without experimentation. Ritter and Wallach (1998) compared an ACT-R model of simple decision making with a projected (but not implemented) Soar approach. R. M. Jones et al. (2007) analyzed theoretical differences between ACT-R and Soar with respect to their procedural memory and decision making implementations, but also did not connect this analysis to experimental model results. Kennedy and Trafton (2007) analyzed procedural learning behaviors between these architectures using the same task and comparable agent designs, but they focused on the scaling of the computational implementations and not on the theory for human cognition and did not compare with human performance. Muller et al. (2008) compared ACT-R and Soar with identical models for a common task and discussed related human cognitive theory at a high level, but their comparison similarly focused on internal program design only and did not discuss how differences in computation led to differences in model task behavior.

Some work by Sun et al. (2004) is closer in spirit to my approach that I describe in the next section. They compared Soar with the CAST architecture for multi-agent communication, for which CAST is specially designed. Each architecture model used the same task knowledge but different computational mechanisms, and their experiment compared the different decision making tendencies of each. Using the same task knowledge allowed computational mechanisms to be more isolated as a variable for evaluation. Their analysis examined computational differences in how each architecture let agents communicate with each other and how this contributed to net decision making patterns. However, they examined net decision making of groups of agents only, and did not explore any other computational facets of these architectures that contributed to their behavior, nor did they vary internal computation elements to determine the effects on external behavioral differences.

Kieras (2016) describes an approach of incremental modifications to agent designs for the purpose of honing in on better models of human processes, using the EPIC architecture. This is also similar to my proposed approach, except that I incrementally compare architectural system processing rather than agent reasoning designs.

In sum, there is not yet agreement on what computational processes provide which human behaviors for tasks, and little uniform precedent in cognitive architecture research for how I might evaluate my computational enactment of PRIMs theory. Therefore, I introduce the novel evaluation approach described in the next section. This approach breaks down the barrier between analyzing external task behavior and the effects of internal processing components. This should allow a greater understanding of the merits of the contrasted architectural theory components, and thereby

contribute toward resolving the discovery problem.

CHAPTER 3

Methodology

For the reasons described above, there are two goals for my methodology: first to develop a comprehensive, task-independent model of human procedural learning that satisfies my four desiderata, and second to evaluate the relative computational and theoretical contributions of my approach in Soar. I aim to accomplish both together through an iterative design and evaluation process.

I begin by reimplementing the model design of Actransfer using Soar, as best as reasonably possible. I then alternate between evaluating model performance and introducing incremental extensions or modifications to the model. This iterative process allows me to experimentally isolate the effects of each change between Actransfer and PROPs in my attempt to make a comprehensive model of PRIMs theory. It also lets me use each evaluation to inform the next incremental modification of the model.

I outline my methodology in Table 3.1 and Table 3.2. Table 3.1 shows my methodology for comparing two different cognitive models that apply the same high-level theory. This approach could be used to evaluate other high-level theories of cognition besides PRIMs and using other architectures besides ACT-R and Soar. The main method steps are described in the center column. The specific application for my work for each step is shown in the right-hand column.

Step	*Method*	*Application*
1.a	Select high-level theory	PRIMs theory
1.b	Define theory flow diagram	See chapter 4
1.c	Select two computational architectures/approaches	ACT-R and Soar
2.a	Implement diagram with 1st approach	Actransfer
2.b	Replicate 1st implementation in 2nd architecture	PROP_1
Iterate until 2nd implementation complete:		
3.a	Evaluate 2nd vs human and 1st, by total and increment	See chapters 7–8
3.b	Revise one part of 2nd implementation toward goal of 2nd approach	See Table 3.2
3.c	Evaluate completed 2nd implementation	See section 9.4

Table 3.1: Methodology steps for comparing architectures

Step	*Method*
3.b.1	Identify gap in model for high-level theory
3.b.2	Identify 2nd architecture solution(s) and select one
3.b.3	Constrain solution to fit model theory and related human theory
3.b.4	Implement revisions to 2nd architecture model

Table 3.2: Methodology sub-steps for extending PRIMs

Table 3.1 includes three main steps. Step 1 and its parts establish the groundwork for the research. Step 2 initializes the actual cognitive modeling. Step 3 iteratively develops the model.

Table 3.1 could be applied to compare any two architecture models that were already defined. However, I use the iteration of step 3 to incrementally develop the PROPs model and thereby extend prior PRIMs work. Table 3.2 shows my methodology for step 3. I now describe each methodology step in more detail.

Step 1.a As described, I select PRIMs theory as the high-level theory for the processing pipeline and principles of task-independent procedural learning, in accordance with D3.

Step 1.b I then formally define a flow diagram of PRIMs theory. This diagram defines the processes that I must comprehensively implement to satisfy D1. I describe my diagram of PRIMs theory in chapter 4.

Step 1.c To compare different computational definitions of the diagram processes, (at least) two architectures must be used to implement it. I specifically compare ACT-R and Soar as two of the most prominent cognitive architectures in the field. ACT-R shares many high-level principles with Soar and other architectures, aligning with the Common Model of Cognition (Laird, Lebiere, et al., 2017), but differs in terms of the computational details it defines and how it defines them. Where ACT-R has been prominent in cognitive modeling, Soar has historically been developed from the context of artificial intelligence and its computational realization, but Soar has been rooted in modeling human cognition as well. I believe this computational focus allows Soar to better define aspects of PRIMs that were elided in its Actransfer application. Finally, there is a great deal of local expertise in Soar, and PRIMs theory has already been applied and tested using ACT-R theory.

PRIMs theory could potentially be applied in other architectures, such as perhaps LIDA (Faghihi & Franklin, 2012), and the comparison between ACT-R and Soar as performed in this thesis could thereby be extended. However, building a unified task-independent system of procedural learning within one architecture is a sufficiently expansive task to make the development of further architectural applications intractable for one thesis.

Step 2.a Once the theoretical groundwork is laid in step 1, the high-level theory must be implemented in both selected architectures. As stated, PRIMs theory has already been implemented in ACT-R, resulting in the Actransfer architecture. This serves as the first architecture implementation. PROPs in Soar is the second.

Step 2.b There are two possible avenues for creating PROPs as a Soar model of PRIMs theory. First is to directly modify the architecture to reflect PRIMs principles. This was the approach used to develop Actransfer from ACT-R. If PRIMs requires a certain type of automatic processing not apparent in Soar, I could add a special component to Soar devoted to that function. A benefit of this approach is that the resulting system would be its own task-independent architecture, and it would properly distinguish its automatic, general processes as architectural. Task-specific instructions given to the model would then also be clearly separated as agent code that runs on the architecture. A downside of the this approach is that architectural modifications would be fairly unconstrained. Overwriting or inserting code in the Soar architecture's definition risks *replacing* aspects of Soar theory with a programmer's naive conceptualization of PRIMs theory. It also tempts over-simplifying process implementation in cases where it might be easy to replicate some desired process output with a custom function rather than actually enact the theorized cognitive processing. This further risks not taking advantage of or investigating avenues that Soar might already provide for enacting PRIMs processing, if overlooked for the sake of expedient development.

The second option is to implement PROPs with production rules that run using unmodified Soar, with the constraint that the rules must be task-independent. A benefit of this approach is that it is constrained to use existing Soar theory. Additionally, it clearly separates the stable Soar theory base from research code. A downside of this approach is that it represents PRIMs processes that are theoretically architectural and automatic as part of the agent's procedural knowledge. This approach also assumes that Soar's automatic mechanisms are already sufficient to comprehensively model PRIMs theory. If this is not the case, I would still need to complement agent programming with architectural modifications. Developing PROPs as a Soar agent then would let me explore whether changes in Soar are indeed necessary to support PRIMs.

I select the second approach for my research. I iteratively develop PROPs as a task-independent production rule agent that runs using existing Soar in a manner that reflects Soar theory. Only if it is shown that PRIMs functionality is not reasonably feasible when using Soar according to its theory and design do I add or modify the underlying code that defines Soar's architecture.

My first implementation of PRIMs theory in Soar is called PROP_1, which largely replicates Actransfer's processing. Where this is inconsistent with Soar processing, PROP_1 simulates Actransfer behavior via agent decision making. For example, where Soar's procedure learning works

differently than ACT-R's automatic pairwise procedure compilation, $PROP_1$ uses agent reasoning to deliberately remember pairs of procedures and set them up for the architecture to compile.

Step 3.a: In this step, I incrementally experiment and introduce changes in PROPs to define more complete PRIMs processing using Soar theory. These incremental additions are reflected in version names: $PROP_1$, $PROP_2$, and $PROP_3$. At each iteration, I evaluate PROPs according to three dimensions:

- How completely does it model every process in the PRIMs flow diagram, including those that were not fully developed in Actransfer?
- How well does it reflect Soar theory?
- How well does it replicate human learning behavior?

The first two evaluations measure the state of PROPs development. PROPs is complete once it both fully models every process of PRIMs and reflects Soar theory. The third evaluation, comparing with human behavior, measures the effects of each processing change during iterative development. The primary measure of interest is whether response time latency for task actions matches human learning and transfer profiles. This also will serve as a final measure of PROPs as a complete model of task-independent procedural learning.

Table 3.1 notes two varieties of evaluation for step 3.a: by total and by increment. For each iteration, I can compare PROPs as a whole with human and Actransfer behavior to determine the *total* results of development up to that point. However, I also want to evaluate the individual changes of each iteration. There are two ways to evaluate *incremental* changes in PROPs. One is to compare the model with its prior iteration. In the first iteration of step 3.a, I would compare $PROP_1$ with Actransfer, then $PROP_2$ with $PROP_1$, and then $PROP_3$ with $PROP_2$. This method is appropriate when an iteration *changes* an aspect of the system design of the previous iteration. However, when an iteration of design instead *adds* a new process detail, it is possible for me to be more precise in evaluating it. I can enable or disable the new system as a hyper-parameter so that I can directly observe the effects on the model's task behavior. This second type of evaluation is preferred wherever possible.

To compare with human learning, as stated, I apply PROPs to the same suite of experiments used to evaluate Actransfer in (Taatgen, 2013), using the same task instructions. By using these different approaches to produce the same task behavior and theoretical cognitive processes, I exploit rather than avoid the uniqueness problem and thereby gain a better understanding of each approach. I particularly examine whether PROPs processing results in superior, worse, or similar fits to human performance compared to the approach of the previous iteration or Actransfer. Wherever results are similar despite different computational approaches, it might indicate a property of

either PRIMs theory itself or of the specific task instructions. Wherever results are different, it must be due to the different model processing.

For both Actransfer and PROPs, I use the same experimental tasks, the same designs of task instructions, the same design of virtual task environments, and the same policies for calculating latencies for environment interactions (such as motor or visual actions). These are all taken from supplementary materials from (Taatgen, 2013).[1] If a difference in the mechanisms of Soar and ACT-R/Actransfer requires that the task instruction designs differ in order to supply equivalent functionality, I make this clear and attempt to make the designs as equivalent as possible.

The chief purpose of having a breadth of experimental tasks during evaluation is to verify that PROPs maintains equivalent functionality and generality as Actransfer in the final model. Using all experiments is not a requirement for intermediate iterations of development. Since each experiment model in the Actransfer suite explains behavior using particular processes, and each iteration of PROPs development modifies different portions of model processing, experiments at each iteration are selected for their relevance in evaluating the modified processing.

Actransfer uses simple time estimates for motor and visual actions in its models, such as 0.25 sec for every eye movement and keyboard stroke. These numbers were not presented as though intended to be particularly precise, but rather to serve as functional approximations. However, because these times do not reduce with procedural learning, they shape the baseline of how fast an agent can perform after all procedural learning is accomplished.

I do not attempt to explore alternate models of task reasoning or motor/visual timing from those provided in the Actransfer experiment suite. The purpose of this research is to fill the gaps in the task-independent PRIMs processes that support this task-specific reasoning and to evaluate the effects of doing so, not to provide better models of task-specific reasoning. I only alter the task reasoning where it is necessary for the PROPs system in Soar to support the overall computation of the original models.

As described above, I at first simulate some of Actransfer's theory in Soar using agent decision making. This exists in PROP_1 and PROP_2, and this decision making takes time during task execution. In general, I can ignore the time for these decisions during post-experiment analysis so that it is as if this functionality had been architectural.

Step 3.b: After each iteration of evaluation, I extend and refine the model further so long as it is not yet complete. In this case, I consider the PROPs system complete when it satisfies all four of my desiderata. PROPs satisfies D1 when it comprehensively models each part of the PRIMs flow diagram. I consider both it and Actransfer to already satisfy D2 as implementable models, since they are by design embedded in architectures that are capable of full environment interaction.

[1] Available at https://www.ai.rug.nl/~niels/actransfer.html

Actransfer similarly already satisfies D3 as a task-independent model, with the exception of the way it uses task-specific timing parameters, as I will discuss in the next chapter. The PROPs system satisfies D3 when it is task-independent in these parameters as well as the ways Actransfer is already task-independent. Finally, PROPs will satisfy D4 as a model of human cognition when each component of its implementation is consistent as a unified cognitive architecture model that aligns with current theory for human cognition and learning.

Step 3.b.1 of my methodology is to identify a gap in the existing cognitive architecture model of PRIMs theory. **Step 3.b.2** is to then examine Soar theory to determine any solution it might offer. It is possible that multiple solutions might be consistent with a proper use of Soar. This might indicate a lack of constraint in Soar theory. In this case, I must still select one to implement in PROPs. I could attempt to select the solution that is also most consistent with PRIMs theory, but thoroughness would demand that I explore both solutions in different iterations of PROPs development, so far as that is tractable. **Step 3.b.3** is to constrain the selected solution to satisfy the various needs of PRIMs theory and human learning theory as well as the existing Soar model. My priority with this step is to maintain a reasonable and consistent model of human learning, and thus satisfy D4. **Step 3.b.4** is to then implement the chosen computation for experimentation.

Step 3.b.3 in particular integrates and unifies Soar theory with PRIMs theory, and the resulting marriage extends both. PRIMs theory contributes to Soar a structure for using its task-independent mechanisms, and Soar contributes a constrained architectural basis for realizing the ideas of PRIMs theory. The model that emerges might include elements not present in either PRIMs or Soar theory alone, thereby extending that prior work. Where neither PRIMs nor Soar defines a process and where their unification implies no further definition, I do not attempt to introduce additional computational explanations for human learning. Rather, I seek to unify existing models.

Step 3.c: Once I complete an iteration of the PROPs system that satisfies all desiderata, the only step that remains is to evaluate this final model. This evaluation process is the same as in step 3.a. After all work has been completed, however, I am able to draw firmer conclusions regarding the implications of any architectural or human learning theory that we use in the development process.

As stated above, I iterate through three versions of the PROPs system: $PROP_1$, $PROP_2$, and $PROP_3$. $PROP_1$ (described in chapter 7) is the initial implementation that replicates basic Actransfer computation using Soar's memory systems. $PROP_2$ (described in chapter 8) modifies this design to incorporate gradual procedural learning using Soar principles. The final iteration of development creates $PROP_3$ (described in chapter 9), which more broadly incorporates Soar theory on goals and decision making. This completes my work in developing the PROPs system. My evaluation of $PROP_3$ constitutes step 3.c for my methodology.

CHAPTER 4

PRIMs and Actransfer

PRIMs theory describes a processing pipeline by which an agent can both execute tasks and learn new procedural rules from primitive "PRIM" rules. Below, I first give a high-level overview of PRIMs theory and its flow diagram before I explain each element in more detail. I then discuss the implementation questions and problems that each phase presents. I then describe the extent to which Actransfer satisfies my desiderata for each phase of the theory, which informs my approach in developing PROPs.

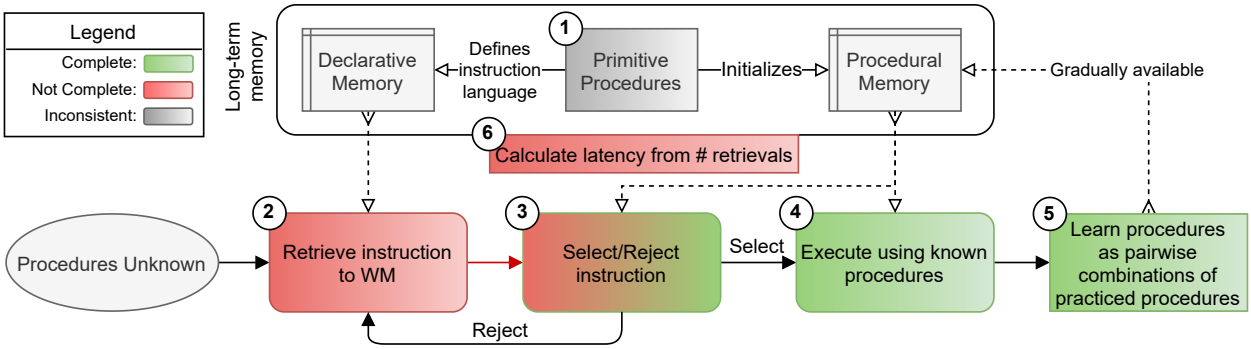


Figure 4.1: The flow diagram of the PRIMs theory procedural learning pipeline. Color indicates completeness of Actransfer's implementation of each phase.

I formally describe PRIMs theory processing with the flow diagram shown in Figure 4.1. The top half of the figure shows declarative and procedural memory stores available to the learner. The lower half depicts the actual procedural learning process. The figure includes numbers 1-6, which mark 6 key phases of PRIMs learning. I will use this numbering to structure my discussion throughout this thesis. I will refer to these phases as P1-P6. In the figure, I color the different phases according to how completely I believe Actransfer implements the theory of each phase in a manner that satisfies my desiderata. Green indicates that the implementation for that phase satisfies all four desiderata. Red indicates that it does not. Gray indicates that the implementation could be considered complete on its own but that it is inconsistent with the human cognitive architecture

theory used for the rest of the implementation and therefore arguably does not satisfy D4. I will explain coloring further in section 4.3.

4.1 Overview

In PRIMs theory, the processing summarized in Figure 4.1 begins anytime the agent does not have automatic procedural knowledge that can apply in its current context. Specifically, the PRIMs process begins anytime the agent does not have rules in its long-term procedural memory that match the current state of Working Memory (WM).

With the PRIMs process, the agent composes condition/action rule behavior for a task from primitive rules (PRIMs). PRIMs make up each condition and action in each composed rule. Initially the agent only knows innate PRIM rules, defined for the agent in the first phase of PRIMs theory, marked P1. The upper half of Figure 4.1 shows P1 and the process of initializing long-term memory. In P2-P5, in the lower half of the figure, the agent compiles condition PRIMs and action PRIMs together to form a single condition/action task rule that it previously did not know as procedural knowledge. Once the agent adds this new rule to its procedural memory, it can then use it to perform the procedure more quickly in the future, as calculated via P6.

In more detail, the agent compiles condition and action PRIM rules together using *declarative* knowledge about a single condition/action task rule, which it retrieves from Long-term Declarative Memory (LTDM) into Working Memory (WM) via P2. The retrieved instruction includes multiple instruction lines for each rule condition and rule action. The agent is assumed to have already been taught declarative instructions that it can use to arrange PRIMs into specific task operations.[1] Once it has retrieved an instruction for PRIMs into WM via P2, the agent uses condition PRIMs in P3 to test whether it should execute the instructed rule actions in P4. In P5, the architecture automatically learns new rules that compile the condition and action PRIMs that the agent used in P3 and/or P4, and it adds these to the agent's procedural memory. This compilation process happens gradually and hierarchically, as I will describe shortly.

Figure 4.2 shows the structure of a declarative instruction that can invoke PRIM rules. From here on, I will use the terminology as depicted. An instruction triggers condition PRIMs with declarative "condition lines" (shown in white) and triggers action PRIMs with "action lines" (shown in gray). The instruction in this figure describes a single task rule as written on the left: "IF (goal==’get-text’) AND (input<>nil), THEN (COPY input to slot1) AND (COPY ‘write-text’ to goal)." In English this could be read, "IF the goal is ‘get-text’ and input is not missing, then copy the input to slot1 in WM and set the goal to ‘write-text’."

[1]The declarative instructions would be created through some unspecified declarative reasoning and comprehension process, such as might be involved in interactive task learning (Laird, Gluck, et al., 2017) with a human instructor.

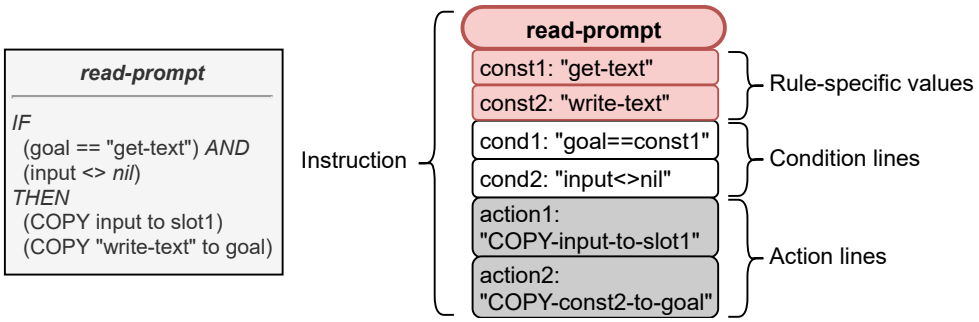


Figure 4.2: An example PRIMs instruction made of condition and action lines.

Notice in the figure that the condition and action lines do not include the values "get-text" and "write-text" explicitly. These are represented separately (as shown in red) in a cache of rule-specific values. This is because PRIM rules for conditions and actions are defined in terms of specific WM operations on specific WM locations. The values "get-text" and "write-text" are not memory locations, but values that can be within memory locations. The first condition line in the figure might more accurately be read, "Test if the value in WM location **goal** is equal to the value in WM location **const1**." When an agent retrieves an instruction into WM via P2, it also retrieves rule-specific values into a reserved set of WM locations so that the agent can use them in general PRIM operations.

In this manner, PRIM rules are like assembly operations in a computer processor that manipulate the values among specific registers in memory. The declarative instructions are the assembly program knowledge by which the agent can invoke these operations to perform any computation supported by the architecture.

The specific types of PRIM operations that could be instructed with condition and action lines depend on the architecture used to implement PRIMs theory. In Taatgen (2013)'s models, most operations can be instructed using only the two basic == and <> types of condition and the single `COPY` action, though each use of a condition or action across different WM locations requires its own unique PRIM rule. For instance, "COPY-input-to-slot1" and "COPY-const2-to-goal" are each applied with a different PRIM rule.

Figure 4.3 shows a simplified example of how phases P1-P6 work in Actransfer's memory systems using instructions as just described. The modeler initializes (P1) procedural memory with PRIM rules that can read single condition lines and action lines. These rules are the same for any task or domain. The figure shows procedural memory with two rules at the top that represent example initial PRIM rules. The modeler can then also initialize LTDM with task instructions that invoke these rules. In the figure, LTDM has three task instructions, labeled "find-prompt," "read-prompt," and "type-word." In this example, these instructions provide operations used in a transcribe-text task. In this task, the agent needs to transcribe text from a prompt into a text editor.

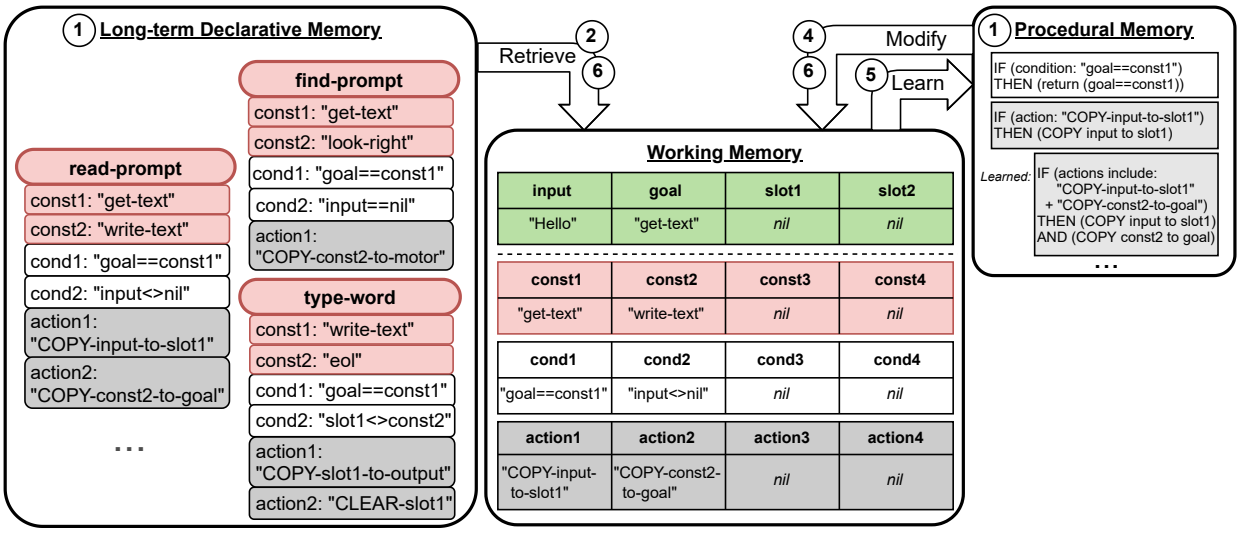


Figure 4.3: A simplified example of Actransfer memory systems and task processing.

If the agent does not yet have text to write, it can attempt to look at the prompt by following the "find-prompt" instruction. After it has done so, it can follow the "read-prompt" instruction to copy what it sees into its WM. Armed with this knowledge, the agent can then begin transcribing the text using the "type-word" instruction.

The figure shows the agent's WM right after it has retrieved (P2) the "read-prompt" instruction. The red, white, and gray boxes show WM *slots* that the agent loads the component elements of an instruction into. Slot labels are shown in bold, and slot contents are shown underneath each label. The agent has a fixed WM capacity, in this example only four slots for each kind of element, though it only needs two of each in this case. The green boxes show general WM slots that are not specific to the retrieved instruction. In detail, the "read-prompt" instruction describes the rule, "If the value of the **goal** slot is "get-text", and the value of the **input** slot is not nil, then copy the value of the **input** slot to the **slot1** slot, and copy the value "write-text" to the **goal** slot."

Once the instruction is in WM, the agent can begin evaluating the condition lines (P3) with PRIM condition rules. This is when the agent decides whether it should perform the instructed actions or not. As shown in the top-right of the figure, the agent has a primitive rule in procedural memory, `IF (condition: "goal==const1") THEN (return (goal==const1))`. This rule matches on the "goal==const1" value in the **cond1** slot of WM, and has the action of telling the architecture whether the contents of the **goal** slot are indeed the same as the contents of the **const1** slot. Another PRIM rule (not shown) will similarly evaluate the "input<>nil" condition line held in the **cond2** slot. The architecture collects the true/false results from these condition PRIM rules for all condition lines in WM.

If all conditions are satisfied, then the agent can also perform the instructed actions (P4). In

this phase, action PRIMs respond to the action lines in WM. The figure shows procedural memory with a rule, `IF (action: "COPY-input-to-slot1") THEN (COPY input to slot1)`. This rule fires, and the contents of the **input** slot ("Hello") get copied into the **slot1** slot. Similarly, another PRIM (not shown) will copy the value "write-text" from **const2** to **goal**.

During this process, the architecture composes new rules (P5) from pairs of PRIMs that fire in sequence. The figure shows one such newly-learned rule at the bottom of procedural memory. The agent would have learned this rule after firing the PRIM action rules for "COPY-input-to-slot1" and "COPY-const2-to-goal" one right after the other in sequence. This new rule can apply these two actions at once for any instruction that includes both of these action lines, even if mixed with additional actions or different const values. This provides the characteristic transfer behavior of PRIMs theory. Since each rule-firing cycle takes time (P6), the agent speeds up task performance by using learned, composite rules in place of multiple primitive rules.

As the new, composite rules execute instructions in future iterations of practice, the architecture further composes pairs of these together into even more specialized new rules. It first composes rules that evaluate condition lines together alongside rules that perform action lines together. With repeated practice, it then learns a rule that performs all condition and action lines together at once. Finally, it learns a rule that performs the entire set of operations at once without the need for an instruction in WM, such as the single rule, `IF (goal=="get-text") AND ("input<>nil") THEN (COPY input to slot1) AND (COPY "write-text" to goal)`. Since declarative retrievals also take time, this further speeds performance. Once the agent has fully learned rules for all instructed procedures, this PRIMs learning process with its declarative instruction retrievals becomes entirely unnecessary.

Figure 4.4 demonstrates how gradual pairwise hierarchical rule learning works in PRIMs theory, using a more complex instruction with two condition lines and four action lines, adapted from (Taatgen, 2013). In the figure, each circle or cluster of circles represents a single rule that the agent uses to apply the instruction line(s) underneath it. At the base of the figure are the individual sequential PRIMs needed to execute each condition and action line.

A PRIMs learning agent learns only one layer of this hierarchy at a time. It is built up gradually over multiple iterations of instruction practice, and with each higher layer of composition, the agent needs fewer rules to execute its instruction lines. After one pass at executing the action lines in Figure 4.4, the PRIMs that execute the first two action lines might be combined into a new composite rule, and the PRIMs for the last two action lines might also be combined in that same pass. The next time the agent needs to apply this instruction, the agent might use just these two new rules to execute all four actions. With that pass, the agent learns a single specialized procedure that carries out all four actions at once. This composition theory comes from ACT-R's model of production rule learning (Ritter et al., 2019).

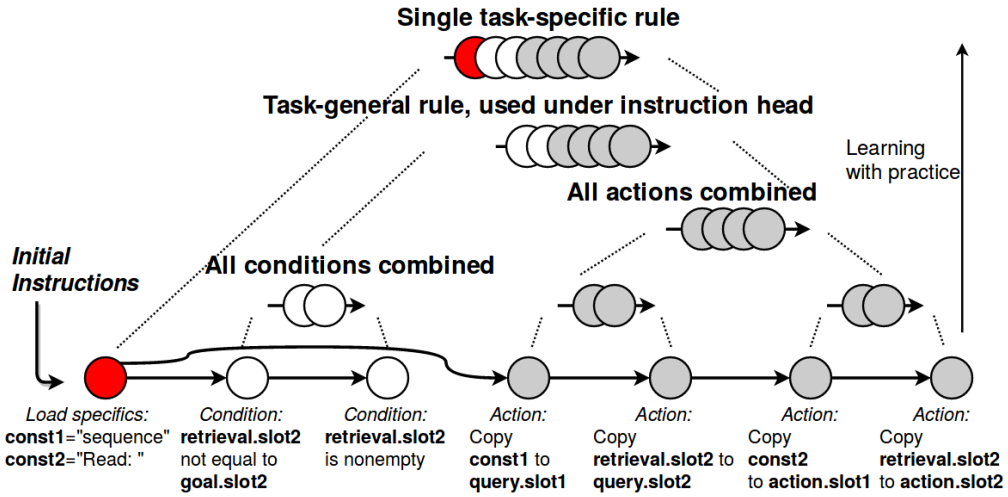


Figure 4.4: Hierarchical clustering of PRIMs procedures with repeated practice, adapted from Taatgen (2013).

As shown at the top of the figure, after all conditions and actions have been combined, the final learning step embeds rule-specific values within the conditions and actions for a single task-specific rule. The agent then uses this rule to perform the task-specific behavior automatically without the need for declarative instruction. The PRIMs process then becomes unnecessary for that operation.

Though the agent performs most quickly by only using the largest compositions of its rules, all rules in the learning hierarchy are available for use in the agent's procedural memory. Even if the agent has fully learned a single task-specific rule for an instruction, it might still need a more primitive set of rules to apply a different instruction. This allows PRIMs theory to model transfer across different task domains. The agent can use the intermediate compositions in the middle of the depicted learning hierarchy when applying a different instruction that uses subsets of the same instruction lines. For example, assume an agent practiced the instruction in Figure 4.4 (call it "I1") enough times to learn a rule for the first pair and a rule for the last pair of action lines, as shown above. If the agent attempts some other instruction ("I2") that includes the same first two action lines as I1, the agent can immediately use its learned rule to perform them both together, even if it had never practiced I2 before. This transfer of memory operation combinations was a major contribution of PRIMs theory.

The net result of this entire PRIMs learning pipeline is that the agent converts declarative descriptions of procedures into more efficient procedural representations through practice.

4.2 PRIMs Phase Details and Challenges

In this section, I describe each PRIMs phase in greater detail along with the particular theoretical and implementation problems associated with each. A model of PRIMs theory that satisfies desideratum D1 (that is, is a comprehensive model of PRIMs theory) must address each of these.

4.2.1 P1: Primitives

PRIM rules are the innate building blocks for creating all other procedural rules that a PRIMs agent learns. Therefore, if they are not properly defined, they significantly constrain which task behaviors the PRIMs agent is capable of. Desideratum D3 requires that the cognitive model be task-independent. When implementing P1 in the context of a cognitive architecture, this means that the innate PRIM rules initialized into the agent must be as task-independent as the underlying architecture. It should be possible to compose any task behavior that the surrounding architecture supports using the set of innate PRIM rules.

An additional challenge for implementing P1 is that the specification of the innate PRIMs defines and constrains the rest of P2-P6. This is because each of those phases depend on PRIM rules in some form. The format of the PRIM rules constrains the format of the declarative instructions, and this constrains the processes that retrieve them and carry them out for task behavior and decision making.

4.2.2 P2: Retrieval Selection

P2 is when the agent selects a procedure instruction from LTDM to retrieve into WM. As I define it here, P2 is not the process of retrieving an instruction but rather the process of selecting a particular instruction from LTDM during a retrieval once that retrieval is initiated. The retrieval itself is initiated by the architecture either when the PRIMs process first begins or when P3 rejects a retrieved instruction.

The primary goal of P2 is to select an instruction that has satisfied condition lines. If the condition lines for the retrieved instruction are not satisfied, the agent will reject the instruction when it gets to P3 and will then attempt another retrieval.

PRIMs theory does not define a particular approach by which P2 should select an instruction from LTDM. The agent's reasoning processes cannot know whether the condition lines in an instruction are satisfied until the agent evaluates them in P3. Taatgen (2013) hypothesizes that a spreading activation mechanism might be used to bias the agent toward retrieving an instruction that would have satisfied conditions in the its current context.

The biggest challenge of P2 is maintaining enough efficiency to support real-time tasks. Some

human tasks, such as many arcade games, can require very rapid real-time responsive behavior on the order of seconds or milliseconds per action. The agent will waste time if it retrieves an instruction with unsatisfied condition lines and needs to reattempt a retrieval. If the agent takes too long searching for an instruction to use, it might not be able to respond quickly enough to perform its task at all.

Desideratum D2 requires that the agent can perform its tasks, while D3 requires that the agent is as task-independent as its architecture in the scope of tasks it can learn, and D4 requires that the agent's processing models human processing. All together, these require P2 to be fast and accurate enough that the agent can support the wide variety of rapid, real-time tasks that humans do, so far as is within the scope of the architecture. P2 retrievals need not be perfectly efficient, just as humans are not perfectly efficient task performers, but they must be efficient enough to permit human-like performance.

4.2.3 P3: Instruction Evaluation

P3 is the phase in which the PRIMs agent evaluates a retrieved instruction and determines whether to apply its action lines or not. During P2, the agent cannot guarantee that the retrieved instruction has satisfied conditions. Thus, once it has retrieved any instruction, the agent first uses known condition PRIMs (or rules that combine condition PRIMs) to test each of the retrieved condition lines against the current state of WM. If any of the condition lines is not satisfied, the agent discards the instruction and repeats P2 to try a different instruction. The more time the agent wastes retrieving and evaluating instructions that are not satisfied, the slower the agent's task performance will be. Once the agent retrieves and evaluates an instruction that does have fully satisfied condition lines it can proceed to P4 to apply their action lines.

P3 is a process in which the agent decides whether it *should* execute the retrieved instruction. This is a form of decision making, and an implementation will thus depend on the underlying architecture's decision making theory. Even if the condition lines for an instruction are satisfied, it is possible the agent might not want to use that instruction at that point in time. There might be other instructions that the agent would be better off applying instead. PRIMs theory does not define any particular constraints for P3, however, other than that it should reject an instruction if any of its condition lines are not satisfied.

The greatest challenge for P3 is D4, which is to maintain consistency with current theory for the human cognitive architecture. In particular, a P3 implementation must be consistent with its architecture's theory for human decision making. Should the agent always choose to apply the first instruction it retrieves that has satisfied conditions? Should it search memory multiple times to see if there are more instructions it could apply instead? If there are multiple instructions it

could choose to apply at a given moment, how many should it retrieve before it decides it has examined enough options to make a sound decision? Because the agent cannot know what choices are available to it until after it retrieves them into WM, these challenges are intimately connected with the implementation of P2.

4.2.4 P4: Procedure Execution

P4 is the phase when the agent uses known PRIM action rules (or rules that combine PRIM action rules) to apply the action lines of the instruction it selected in P3. Action rules are special in that they do not require any task-specific conditions. They only need to test whether the agent in P3 chose to apply the instruction, plus of course the presence of the specific action lines in WM.

P4 processing is responsible for selecting which known rules the agent should use to apply its action lines. At first, the agent will only have PRIM action rules to choose from. However, as the agent learns combinations of PRIM rules in P5, it will have more and more rules at its disposal. And what the agent learns in P5 will depend on which rules the agent uses in P4, and this will largely define the agent's ability to transfer procedural knowledge. The processing needs of P4 are thus intimately connected with P5.

Depending on how P3 is implemented, the connection with P5 applies to the use of condition PRIMs in P3 as well. The original publication of PRIMs theory assumes this is the case (Taatgen, 2013). However, as I discuss in chapter 9, this need not necessarily be true.

While the agent *can* always use PRIM rules to apply a selected instruction, effective learning in PRIMs theory requires that the agent transition to using learned rules. Otherwise, there is little advantage from the PRIMs learning process.

The main challenge for implementing P4 is D4. The implementation of P4 will have a significant impact on shaping the model's task performance and learning curve, and thus whether these will be similar to human learning or not.

4.2.5 P5: Procedure Combination

P5 is the process of gradually compiling practiced rules together for faster future execution. PRIMs theory requires an architectural mechanism that compiles pairs of practiced rules automatically. This should result in a binary hierarchy of composed rules over time, as depicted in Figure 4.4.

P5 differs from P2-P4 in that it is an automatic background architectural process invoked during the agent processing in P2-P4. PRIMs theory assumes that the underlying architecture provides an automatic procedural learning mechanism, as is the case in both ACT-R and Soar.

It is important in PRIMs theory that the agent compose conditions and actions separately. If the agent applies any action lines at the same time that it tests only a subset of the condition lines,

it may find that the remaining condition lines are not satisfied. Then it would need to undo the actions.

For the context of this thesis, I am working with the Actransfer and Soar cognitive architectures. Each of these already defines its own automatic procedural learning mechanism (Actransfer's comes from ACT-R). These mechanisms are also already task-independent in their design, and thus already satisfy D3. Therefore, my main concern for P5 is D4, specifically that the agent use the architecture's learning mechanism in a manner consistent with the architecture's theory and human learning theory in general.

4.2.6 P6: Latency

P6 is the process of calculating the time cost of P2-P5, specifically for modeleing human task performance. Without a specific method for translating agent processing to task reaction time, there would be no way to measure if the model can explain how humans improve in reaction time with procedural learning and transfer. In PRIMs theory, there are two specific assumptions that define P6. First is that it takes the same amount of time to use a PRIM rule as a rule that is composed of two or more PRIMs. This means that an agent takes less time to perform its task operations when it uses learned composite rules in place of many separate primitive rules. Second is that it takes time to retrieve an instruction from LTDM. PRIMs processing always requires at least one instruction retrieval. This means that an agent also takes less time when it uses fully-proceduralized rules (such as shown at the top in Figure 4.4) for its task operations in place of PRIMs processing.

PRIMs theory does not, however, define further details for P6, such as how much time to allot per procedural or declarative retrieval or what other parameters, if any, shape the latency of PRIMs processing. In my case, these details come from the theory of the underlying cognitive architecture used in the implementation. Taatgen (2013) used the timing theory of ACT-R to define P6 in Actransfer.

P6 is important especially for desiderata D3 and D4. To satisfy D3 for task-independence, the method of calculating latency from procedural learning should be common across tasks. To satisfy D4, the way that the agent demonstrates procedural learning should also be consistent with the way the underlying cognitive architecture is intended to be used as a model of human cognition. For instance, the model of procedural learning should not be primarily based on an architectural function for getting faster at perceptual processing, but it should use the architecture's function for getting faster at procedural processing.

4.2.7 Summary

In summary, each of these phases pose the following challenges:

1. **Primitives**: What initial set of primitive procedures supports truly task-independent behavior?
2. **Retrieval selection**: How does the agent select a specific instruction when retrieving from LTDM?
3. **Instruction evaluation**: How does the agent evaluate and choose to apply a retrieved instruction with decision making?
4. **Procedure execution**: How does the agent use known procedures to execute a selected instruction?
5. **Procedure combination**: How does the agent compose practiced procedures into new procedures?
6. **Latency**: How does the agent's cognitive processing with PRIMs map to temporal costs in task behavior to allow comparison with humans?

In later chapters I present the PROPs system as a theory and model that comprehensively addresses each of these phases in a way that satisfies my desiderata.

4.3 Actransfer's Completeness for Implementing PRIMs

In the rest of this chapter, I briefly describe how the Actransfer implementation of PRIMs theory does or does not satisfy my desiderata for each of P1-P6. This establishes where PROPs needs to focus to supply a comprehensive implementation of PRIMs theory. I do not attempt to provide a full description of how Actransfer functions. For a more in-depth description of Actransfer's approach, see (Taatgen, 2013).

As mentioned earlier, color in Figure 4.1 indicates the degree to which I believe Actransfer's implementation satisfies my desiderata for each of P1-P6. Specifically, green indicates complete satisfaction. Gray indicates completeness that might be inconsistent with architectural theory. Red indicates processing that produces the desired task functionality but does not model the cognitive processing as constrained by my desiderata. Based on my analysis, Actransfer's computation satisfies my desiderata for P3, P4 and P5, but not for P1, P2, and P6.

4.3.1 P1

Consistent with ACT-R theory, Actransfer defines PRIMs as production rules in procedural memory, where the architecture selects one rule to fire per decision cycle.

Each PRIM rule in Actransfer is defined as one specific assembly operation on specific WM slots. Actransfer requires a different innate PRIM rule for each possible condition or action line that uses different slots. For example, it would need one PRIM rule for the action "COPY-input-to-slot1" and another PRIM rule for the action "COPY-input-to-slot2," and so on for all permutations of slots. The agent's procedural memory is initialized with the full set of possible permutations of memory operations across all slots. The total number of PRIM rules depends on the number of WM slots. In Actransfer, this is an architectural parameter. When Taatgen configured Actransfer with 31 WM slots, this resulted in 1,693 PRIMs for the combinations of these slots with each type of operation (Taatgen, 2013).[2] Figure 4.5 shows the Actransfer WM model and the relation of PRIMs to individual WM slots. The white circle and arrow represents a specific condition PRIM rule that can compare the values of two slots. The gray circle and arrow represents a specific action PRIM that can copy the value from one slot to another.

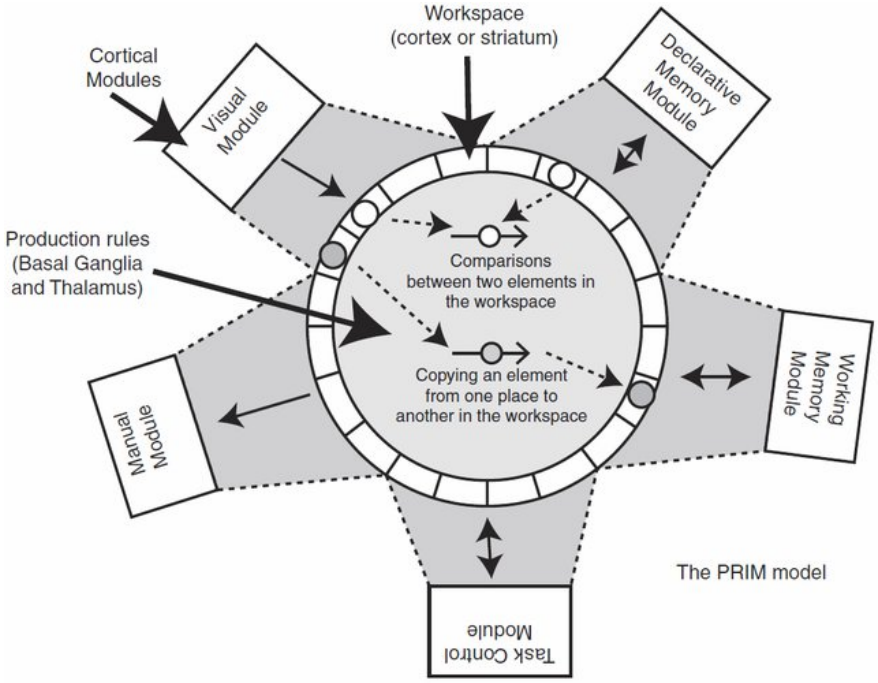


Figure 4.5: The Actransfer cognitive architecture memory model, taken from (Taatgen, 2013).

In order to initialize these permutations of PRIMs across all WM slots, Actransfer added a constraint to ACT-R's use of WM. ACT-R organizes WM into a number of buffers, each of which

[2]The choice of 31 slots is arbitrary, but was chosen by Taatgen as sufficient to support the processing needed for his task models without entailing a prohibitively large burden to memory or processing.

interface with a specific module of the architecture. Figure 4.5 shows five ACT-R modules also used in Actransfer, such as the "Visual Module," the "Declarative Memory Module," and so on. In ACT-R, each WM buffer for each module is capable of holding a single declarative *chunk.* An ACT-R chunk is a single declarative object made of any number of slot elements. A buffer might first hold a chunk that contains 4 slot elements, and then the agent's decision making might replace that chunk with a different chunk that contains 10 slots. Actransfer changed this model so that each buffer is constrained to have a fixed small number of slots.[3] The figure depicts each buffer with 4 slots.

This restriction deviates slightly from ACT-R theory by constraining the way declarative knowledge must fit into WM. At first it might seem to be an inconsequential modification, but it could have significant implications on agent theory and function. Actransfer's constrained WM capacity constrains all the agent's LTDM representations, because a long-term memory must be retrieved into WM to be useful. Actransfer models are as a whole constrained to only use algorithms that can work with the given limited WM chunk capacity. In ACT-R models, by contrast, WM chunk capacity is constrained by the agent's cognitive processing and by how the agent forms and uses knowledge for its context. As a computational model, Actransfer's design also inserts the strong assumption that the configuration of possible WM slots within buffers is fixed and innate in human cognition, while ACT-R only entails the assumption that the number of buffers is fixed and innate while the content in them can change over time. Actransfer also requires that procedural memory is initialized with PRIMS, whose number scales combinatorially with the number of configured WM slots.

It is doubtful that the computation of Actransfer was meant to present a particular stance in the human memory debate rather than to simply allow a functional model of PRIMs learning, and Taatgen (2013) did show that Actransfer computation was sufficient to demonstrate PRIMs theory. It is also possible that, for most practical purposes, this design choice for Actransfer merely constrains ACT-R's usage without sacrificing its overall capabilities. Nevertheless, I note that it does represent a change from ACT-R theory that constrains the representation of task knowledge and the content of PRIMs instructions. For this reason, while it might be considered to satisfy D1 as a comprehensive implementation of PRIMs theory, and while it might also perhaps satisfy D4 as a model of a theory of human WM with respect to only P1, this could be considered to not satisfy D4 with respect to consistency with the whole architectural model. P1 is therefore marked in gray in Figure 4.1.

[3] Architecture parameters set the exact number of slots for each buffer.

4.3.2 P2

Actransfer relies on ACT-R's declarative activation mechanism to support P2, the process of selecting an instruction during a LTDM retrieval. However, the architecture uses oracle knowledge of which instructions have satisfied conditions to artificially bias the agent toward retrieving those instructions. I do not consider this to represent a sufficient model of human cognitive processing for the purposes of my desiderata.

As stated earlier, Taatgen (2013) hypothesized that some sort of spreading activation among memories, such as ACT-R supports, might achieve the behavior PRIMs requires for instruction memories. That is, declarative knowledge about the task and the world state would be associated with long-term instruction memories in some way so that instructions with satisfied conditions would have priority for retrieval over non-satisfied instructions. But this was not implemented in the published account of Actransfer. Instead, Actransfer introduced an automatic routine within the ACT-R system so that, whenever the agent attempts to retrieve a declarative instruction, the architecture performs a brute-force evaluation of all conditions of all instructions in LTDM, testing whether they match current WM. Then the architecture increases the activation of instructions with satisfied conditions. The ACT-R retrieval mechanism retrieves a single chunk from LTDM, with priority to memories with the highest activation. (moderated with some random noise). Thus, since this added routine in Actransfer ensures that instructions with satisfied conditions will have high activation, the agent will almost always retrieve an instruction with satisfied conditions.

This computation is sufficient to allow the agent to function, but it does not perform the hypothesized spreading activation for P2, and thus it does not satisfy D1 for comprehensive modeling. Actransfer's approach for P2 is therefore marked in red.

4.3.3 P3

In Figure 4.1, P3 is shaded both red and green, with red coming from a red arrow between P2 and P3. In PRIMs theory, P2 and P3 are intimately connected. While Actransfer's implementation of P3 satisfies my desiderata, it cannot function satisfactorily due to the limited options passed to P3 from P2.

Actransfer uses ACT-R's long-established rule selection and firing mechanism to test instructed condition lines, via PRIM rules and their compositions. In ACT-R's design, the architecture associates a utility value with each rule in procedural memory, and when multiple rules in procedural memory have satisfied conditions, the architecture biases the agent to select the one with the highest utility. This is conceptually similar to the way that the agent is biased to retrieve a declarative instruction with the highest activation, but they each function differently. Activation is altered by how often an agent uses a declarative memory and by spreading activation. Utility is altered by

how often an agent uses a procedural rule and by environment reward, as well as by the procedural learning mechanism that I will describe shortly. The architecture also updates activation and utility using different formulas according to what has been shown to align with human behavior for each process.

Actransfer uses this mechanism to select which rules it uses to evaluate retrieved condition lines in P3. For any instruction that the agent retrieves via P2, the agent uses one or more decision cycles to fire condition rules that evaluate the retrieved condition lines. Like ACT-R, Actransfer requires one decision cycle per rule.

Inconsistent with ACT-R theory, Actransfer does not use this rule selection and firing mechanism to select from among competing behaviors for the actual task. This is because Actransfer's P2 only ever provides one choice to P3 at a time, and P3 immediately selects it for execution if its condition lines are satisfied. If the agent has multiple instructions it could follow, which are equally valid and have satisfied conditions, the agent's declarative retrieval process determines which one to use and not its decision making process. Thus, P2 retrieval selection is the locus of task decision making, instead of P3 or the architectural rule selection mechanism. This is counter to ACT-R theory.

Thus, while Actransfer uses ACT-R theory properly to moderate declarative retrievals in P2 as well as to evaluate conditions in P3, it goes against ACT-R theory to the extent that it uses P2 retrieval selection to replace choice-based decision making. This means it does not satisfy D4 for my work as a consistent architecture model of human cognition.

4.3.4 P4

Actransfer uses the same utility-based rule selection and firing mechanism to apply action lines as it uses to evaluate condition lines. As was the case for P3, this decision making process in ACT-R is a reasonable model of PRIMs theory for P4.

ACT-R's utility-based rule selection process provides the solution by which the Actransfer agent selects whether it uses primitive rules or learned composite rules to apply instruction lines (for both action lines in P4 and condition lines in P3). At first, a learned rule that combines two other rules might not have a high utility, and the agent might not be likely to use it. However, each time the agent practices the two component rules together in sequence again, the architecture increases the utility for the learned rule that combines them. Thus, learned rules gradually increase in utility until they are used in place of their component rules. This also substantiates PRIMs processing without notable deviation from ACT-R theory, and satisfies D4 and the other desiderata. P4 is therefore marked in green.

4.3.5 P5

Actransfer uses ACT-R's procedure compilation mechanism to learn new rules. ACT-R attempts to combine[4] any two rules used in two sequential decisions into a single new rule. This provides the hierarchical pairwise hierarchy of PRIM composition depicted in Figure 4.4 (and is integral in the design of PRIMs theory to begin with). This substantiates PRIMs processing according to ACT-R's well-supported procedure compilation mechanism, and this satisfies D4 and the other desiderata. Thus, P5 is marked in green.

4.3.6 P6

Actransfer models calculate latency for cognitive processing as a mix of declarative retrieval and decision cycle times, using the same approach for these as ACT-R. Each declarative retrieval is simulated to take a variable amount of time, according to a formula I will describe shortly. Each decision cycle corresponds to the use of a single rule from procedural memory, and is simulated as requiring 50 msec. Actransfer models also include time for motor actions and perception in their predictions for total task reaction times, but these are outside the scope of cognitive processing that PRIMs theory describes and must be added separately.

ACT-R calculates latency for a declarative retrieval according to the activation of the retrieved memory, where more highly-activated memories take less time to retrieve. The exact formula is:

$$T_{retrieve} = F_r \times e^{-A} \tag{4.1}$$

where A is the activation of the retrieved declarative chunk, and F_r is a *latency factor* parameter that can vary from model to model (Brasoveanu, 2015). The default value is $F_r = 1.0$.

The architecture adjusts activations for long-term declarative memories over time as those memories are used. This represents a *declarative learning* process in ACT-R. A memory's activation increases from being retrieved frequently into WM, and thus its retrieval time decreases with repeated access. Other factors such as association with other activated memories in WM or perception (spreading activation) also increase a memory's activation. A memory's activation decreases over time. Equation 4.1 means that Actransfer models an instruction retrieval as requiring less time the more the instruction is used.

In Actransfer, procedural learning also gradually reduces the *number* of declarative instruction retrievals. The agent retrieves condition and action lines one at a time right before it uses a rule to perform each, and each gets its own added $T_{retrieve}$. Task-specific constants also require a separate retrieval. Thus, an agent that has not yet learned any rules would use 8 declarative retrievals to

[4]Not every pair of rules can be combined. For example, two rules in which the first initiates a LTDM query and the second reacts to the query result cannot be combined while preserving the desired functionality.

invoke a declarative instruction made up of 3 condition lines and 4 action lines, one for each condition and action line plus one at the start for the constants. Overall, while this makes declarative retrieval time tied to procedural learning, $T_{retrieve}$ is still declarative retrieval time and theoretically distinct from procedural learning time.

As stated, PRIMs theory describes procedural learning and transfer as a result of the procedural learning process of P5. But for some Actransfer models, most of the decrease in latency is due to the *declarative learning* process that comes from increasing declarative memory activation, separate from the results of P5 learning. This is demonstrated in the next section. Furthermore, in order to achieve fits to human data for different tasks, the value of F_r in some cases differs by an order of magnitude for each task. Tuning F_r to fit human profiles for different tasks is common in ACT-R models (Brasoveanu, 2015), but it does not align with D3 for my work. Taatgen (2013) does not mention declarative activation effects as part of PRIMs theory. Though PRIMs theory requires declarative retrievals for the procedural learning process, declarative learning and procedural learning are separate learning processes. ACT-R theory also distinguishes that one should use decision-cycle latency to model procedural learning and declarative activation to model declarative learning (Anderson et al., 2019). Thus, to the extent that declarative learning replaces procedural learning in Actransfer as its computational model of human procedural learning, Actransfer deviates from PRIMs theory and does not satisfy D1. P6 is therefore marked in red in Figure 4.1.

CHAPTER 5

Actransfer Experimentation

I now describe the Actransfer experiments and results from (Taatgen, 2013). These demonstrate PRIMs theory and Actransfer as just described and supply the context for my iterative development of PROPs. I then describe my PROPs implementation in contrast to Actransfer in chapters 7-9.

Taatgen (2013) demonstrated his theory and its applicability for modeling human transfer using a suite of 4 human experiments, involving 6 tasks. These are the experiments I also use for evaluating PROPs, as follows:

1. **(Elio, 1986)**: Mental arithmetic task
2. **(Singley & Anderson, 1985)**: Text editors task
3. **(Chein & Morrison, 2010)**: Complex Verbal WM task, Stroop task
4. **(Karbach & Kray, 2009)**: Task Switching task, Count Span task, Stroop task

I will briefly review each experiment and the modeling results from Actransfer. These are the results to which I compare my iterations of PROP development.

5.1 Mental Arithmetic Task

Step	Calculation	Op Type
1: Particulate rating	$\text{Solid} \times (\text{lime}_4 - \text{lime}_2)$	Component
2: Mineral rating	greater of $(\text{algea}/2)(\text{solid}/3)$	Component
3: Index 1	$\text{Particulate} + \text{Mineral}$	Integrative
4: Marine hazard	$(\text{toxin}_{\text{max}} + \text{toxin}_{\text{min}})/2$	Component
5: Index 2	$\text{Index1}/\text{Marine}$	Integrative
6: Overall quality	$\text{Index2} - \text{Mineral}$	Integrative

Figure 5.1: Example arithmetic task procedure. Component steps only reference inputs. Integrative steps require remembering results of previous calculations.

SOLID	ALGAE	LIME	TOXIN
6	2	3	4
		5	8
		1	7
		9	2

Figure 5.2: Example arithmetic task inputs. Participants look up hypothetical water sample data from among ten values provided per trial. For lime or toxin values, procedure directions either specify the row index to look up or instruct to find the max or min value.

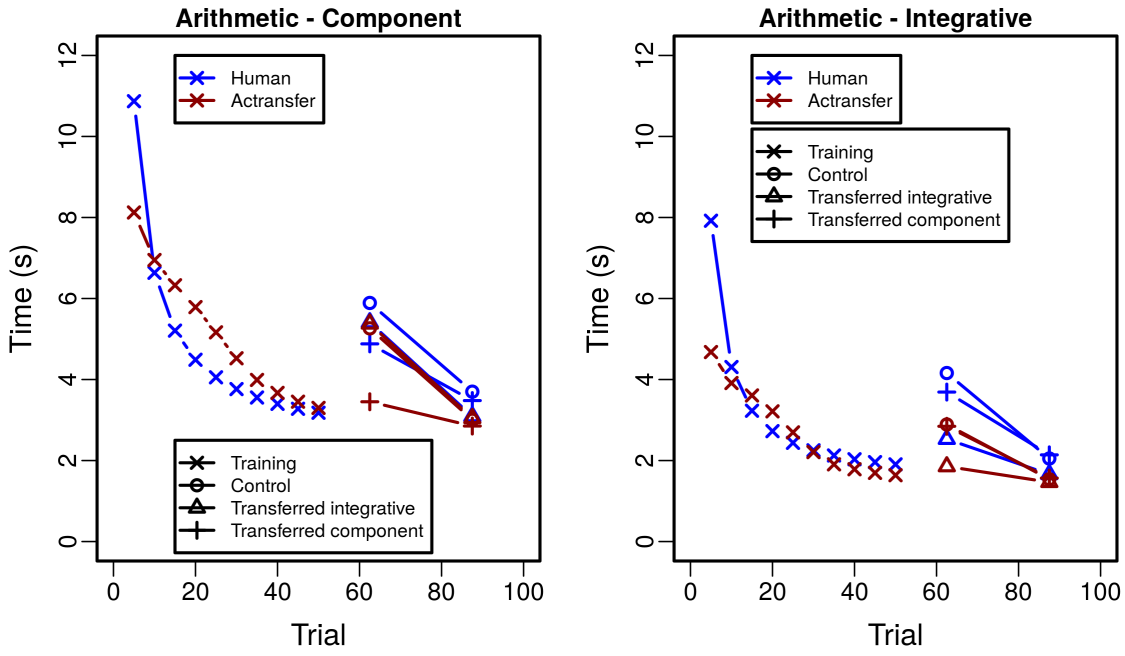
The mental arithmetic task involved calculating hypothetical pollution rates based on water samples. Subjects repeatedly performed mental calculations using given input values. In the human study (Elio, 1986), subjects were trained in an initial procedure until they achieved perfect recall, and were then tasked with performing it 50 times on various inputs. Following this, subjects were assigned to 50 trials of one of three transfer conditions: transferred integrative, transferred component, and a control. The first two of these were different math procedures that shared different types of calculations with the training, either the integrative steps, which were steps that required remembering results of other steps, or component steps, which did not. The control shared no calculations with training.

Figure 5.3 shows human performance in blue. The first 50 trials show the reported power-law fit to performance in training. Trials 51-100 show reported data for the three transfer conditions. Elio (1986) reported transfer data as the mean from the first and last 25 trials per subject.[1] In the original study, results for component and integrative calculations were reported separately. I show data for component steps in Figure 5.3a and for integrative steps in Figure 5.3b.

A basic ACT-like identical productions model would predict transfer from the training procedure to procedures that shared calculations, but would not predict transfer to the control. Yet transfer to the control was evident in the human results.

Figure 5.3 shows Actransfer model performance in red. Results are the average of 8 experiment repetitions. The key result is that this model produces the same relative transfer trends in the control case. The agent also displays a higher percentage of transfer for the *transferred component* case in Figure 5.3a and for the *transferred integrative* case in Figure 5.3b. These cases respectively share component or integrative calculations with the training, and this allows classic identical productions transfer in addition to primitive instruction line transfer. The Actransfer model overall is not a perfect fit to the human plot, but it shows the same transfer trends. I show no error bars for the Actransfer model data because standard error is too small to be visually meaningful (about 0.01 sec for most data points).

[1]The original 1986 human data is not available - only the data shown (R. Elio, personal communication, May 18, 2018).



(a) Component steps

(b) Integrative steps

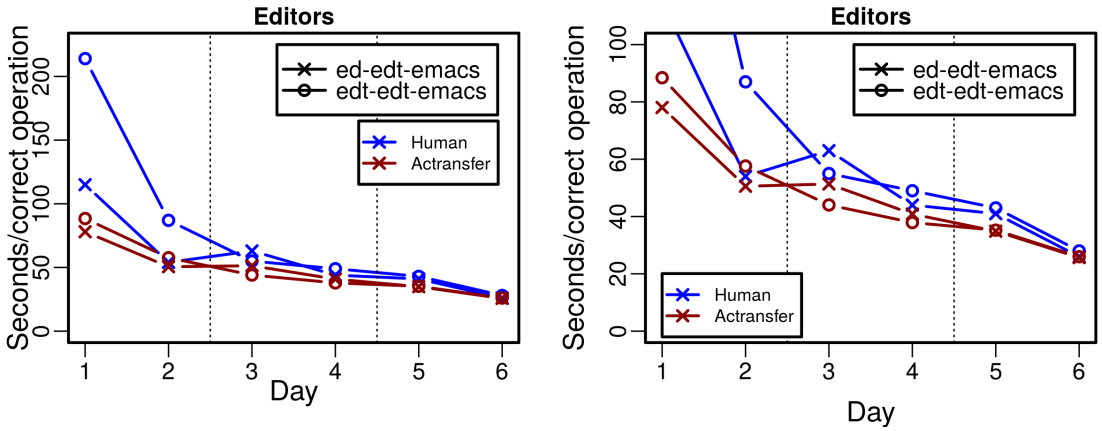
Figure 5.3: Human and Actransfer performance. Data for problems 1-50 show training performance. Data for problems 50-100 show performance for each of the three transfer conditions. Actransfer data were generated using supplementary materials from Taatgen (2013).

5.2 Editors Task

In the editors task (Singley & Anderson, 1985), typists modified documents according to written edit directions. Example directions included replacing one word with another or deleting a sentence. Three keyboard-only editors were used with which participants had no prior experience: ED, EDT, and EMACS. These each require 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, call this case ED-EDT-EMACS. If EDT performance was faster after using ED than when using EDT on day one, this indicated transfer to EDT.

For brevity, Figure 5.4 shows only human and Actransfer performance when transferring from ED to EDT-EMACS, as other editor results are comparable. The figure on the left is scaled to show all data points. The figure on the right shows the same data at a smaller scale to better see the details on days 3-6. Model results are the average of 12 experiment repetitions. No error bars were provided in the original human study, and standard error for the Actransfer model is less than 1 sec and not visible at the scale in these figures. Again, human performance is shown in blue, and



(a) Full scale

(b) Half scale

Figure 5.4: Human data from (Singley & Anderson, 1985) and the Actransfer model from (Taatgen, 2013), demonstrating transfer between editors.

Actransfer model performance is shown in red. Humans are almost as fast at EDT after two days of ED as after two days of EDT, indicating substantial transfer. They show similarly significant transfer to EMACS on day five. (EMACS-only users required about 80 sec on day 1, not shown). The Actransfer model is faster than human performance during days 1-2, but the chief result of this model is that transfer trends are again the same.

5.2.1 Timing Methodology

The editors and arithmetic experiments demonstrate Actransfer's application of P6 described in the previous section. As stated, Actransfer uses Equation 4.1 ($T_{retrieve} = F_r \times e^{-A}$) to calculate how long each memory retrieval takes, with a task-specific latency scaling factor F_r. Because the effect of this parameter for a model depends on the way memory activations change within the specific model, I re-ran the Actransfer model for the editors and arithmetic tasks and measured the time for each retrieval. For the original editors task, F_r was set to 1.5. Combined with the model's use of memory, this meant early instruction retrievals took about 0.95 seconds per operation, and gradually reduced to about 0.3 seconds by the end of day 6, as activation for task-relevant instructions increased. By contrast, for the arithmetic task model F_r was set to 0.15. Arithmetic task retrievals decreased from 0.07 to 0.03 seconds over the course of that task.

To better evaluate the effects of this F_r scaling in Actransfer modeling, I also re-ran the Actransfer agents for both tasks with F_r set to 0, so that LTDM retrievals took no simulated time. Results are shown in Figure 5.5 and Figure 5.6. Figure 5.5 shows clearly that most of the editors agent's timescale is from $T_{retrieve}$. Figure 5.6 shows that $T_{retrieve}$ plays a less significant role in the arithmetic task, since F_r was set to a low value from the beginning, but it is still appreciable. By

itself this result would not necessarily mean that most of the editors learning was declarative learning, since the number of declarative retrievals reduces with procedural learning. But, as stated, each individual retrieval in the editors task reduced from about 0.95 sec to 0.3 sec over the course of the task, more than a two-thirds change. Thus, more than two-thirds of the difference between Figure 5.4 and Figure 5.5 is from declarative learning specifically, not procedural learning.

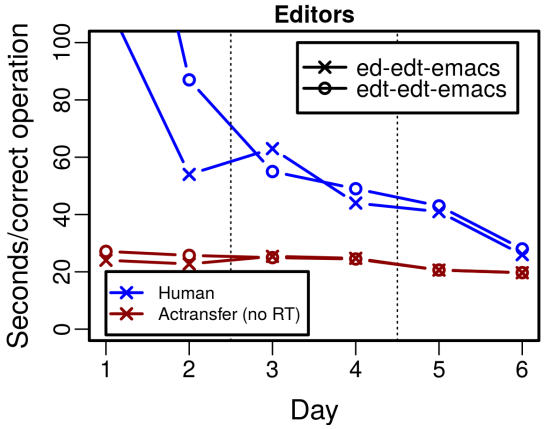
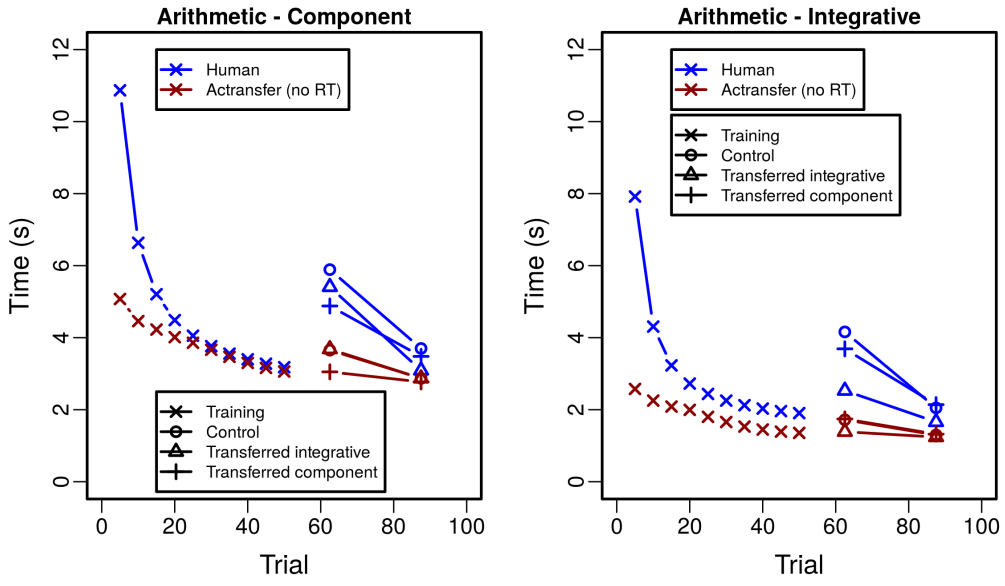


Figure 5.5: Actransfer agent for the editors task, without declarative retrieval latency.



(a) Component steps

(b) Integrative steps

Figure 5.6: Actransfer agent for the arithmetic task, without declarative retrieval latency.

These tasks are designed to predominantly test the subjects' procedural rather than their declarative abilities in the sense that they test how subjects learn to perform routines rather than to report

memorized facts. Nevertheless, it is possible that subjects in the editors task might require significantly more declarative effort than for the arithmetic task, due to the need to remember different keyboard commands. The higher F_r might represent this. However, the free-parameter nature of F_r highlights the need for a more precise computational definition of how this latency arises, and why there is such a difference in scale between these tasks. The dependence of the models upon F_r to demonstrate human performance shows that something significant is missing from the model computation.

5.3 WM Training and Stroop Experiment

In the (Chein & Morrison, 2010) experiment, subjects were trained for 20 days on a complex WM task. As part of this task, subjects were sequentially shown a series of single letters on the screen, and then asked to report the sequence. The sequence would increase or decrease in length (and difficulty) if subjects consistently remembered the list accurately or inaccurately. Between each letter, subjects were given a distractor task of identifying a word (e.g. "blick") as real or not.

Before and after this training, subjects were given a battery of cognitive assessments to measure how training transferred to other cognitive abilities. Control subjects were assessed before and after 20 days of no training. One assessment that showed surprising results was the Stroop task, in which a single word describing a color (e.g. "red") was shown in a font of a potentially different color (e.g. blue), and subjects were asked to report the color of the font. A series of such prompts were shown to participants, with small pauses between each trial. The assessment measured Stroop Interference: the difference in average response time from when text and font colors are *incongruent* (e.g. "red" in blue font) and *congruent* (e.g. "red" in red font).

Figure 5.7 shows human performance for this Stroop assessment in blue. In the figure, interference indicates the *difference* in average response time between the incongruent and congruent Stroop words. The horizontal axis shows the change in interference before and after the 20 day WM training period. Results showed surprisingly significant transfer from WM practice toward reduction in Stroop interference.

The Actransfer model of this experiment explained this as transferred decision making, specifically the decision to prepare for task stimuli in between trials. During pauses in the training task, the model prepared by rehearsing the sequence of known letters. During pauses in the Stroop task, the model could choose to either be idle or to prepare to focus on font color. In the model, the instructions for the prepare operation were composed of the same PRIMs in both tasks. This meant that both instructions increased in activation during practice. This higher activation biased the agent to retrieve the prepare instruction in the Stroop task.

Actransfer generated Stroop interference data via LTDM retrieval latency, based on Equa-

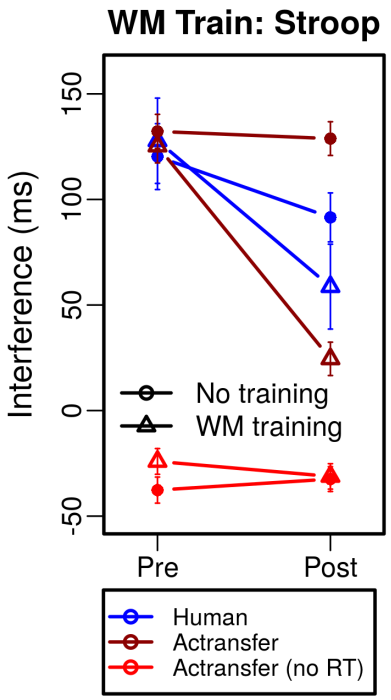


Figure 5.7: Interference in the Stroop task, before / after WM training. Human data are from Chein and Morrison (2010) and Actransfer data from Taatgen (2013).

tion 4.1. The contents of the agent's visual buffer in WM spread activation to associated declarative memories in LTDM. If the agent had conflicting stimuli in the its visual buffer in WM (the written color word and the font color were incongruent), this spread was diluted across memories for both colors. If the agent chose to prepare, however, it would choose to only allow the presented font color to be loaded into its visual buffer. In that case, only that font color's declarative memory received spreading activation. This resulted in higher activation and therefore a faster retrieval.

Figure 5.7 shows Actransfer results in dark red. Results are the average of 9 experiment repetitions. This model of transferred decision making produces substantial reduction in interference with training. Performance is again not an exact match to human data, but it shows the same overall transfer trends.[2]

I also ran an Actransfer model with F_r set to 0. These results are shown in bright red in the figure (`(no RT)` for no retrieval time), but they have little meaning because the model defines interference using $T_{retrieve}$. Without $T_{retrieve}$, the model demonstrates negative interference regardless of training. That is, the model is slightly faster for incongruent prompts than for congruent prompts.

[2]Actransfer was also used to model human subjects' improved declarative memory recall in the WM training task over the course of training. I ignore this model here because it demonstrates the ACT-R declarative learning mechanism, which is already well established, rather than procedural learning with PRIMs.

This experiment demonstrated how the use of PRIMs can transfer decision making, given how P2 determines decision making in Actransfer. A consideration for my work with PROPs is whether this same transfer holds when P2 is implemented to reflect both PRIMs and Soar theories.

5.4 Task-switching Experiment

The task-switching experiment from (Karbach & Kray, 2009) tested whether training in task switching within one set of tasks transfers to task switching with a different set of tasks, as well as to performance in various cognitive test tasks. Similar to the Chein and Morrison experiment, human subjects were given a battery of cognitive tests before and after training. This battery included a task-switching task (which differed from the training task-switching task) and a Stroop test.[3]

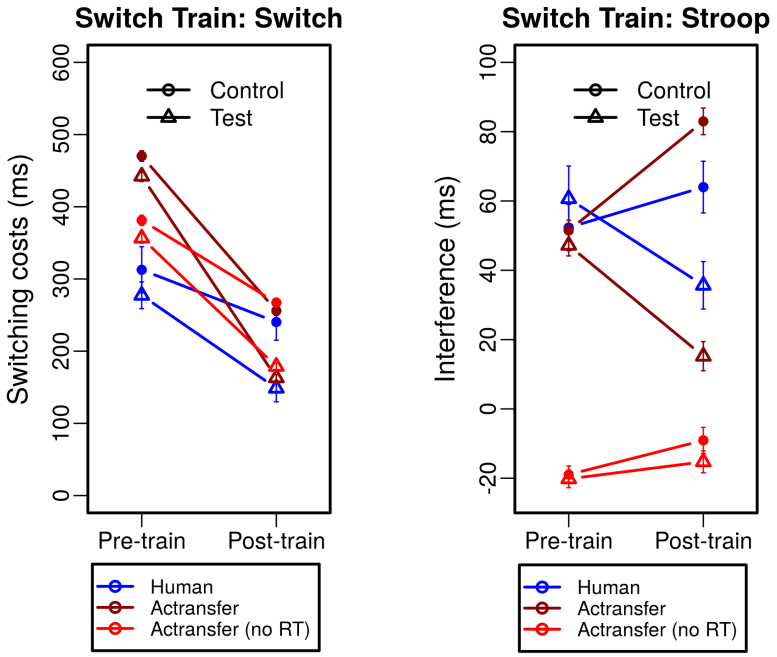
In the training, subjects were iteratively shown different images containing either one or two planes or cars. Subjects had to switch between two different response tasks, given the same types of images. The first task was to report whether the image showed planes or cars. The second was to report whether there were one or two items shown in the image. Subjects were trained in blocks of just one of these two tasks and in blocks of switching between these two tasks every second trial. Training took place over the course of four days, each of which involved 8 single-task and 12 task switching blocks. Control subjects, however, trained only in single-task blocks.

In the pre- and post-tests, the alternate task-switching task was like the training task, except pictures were of vegetables and fruit rather than cars and planes, and subjects reported whether these were small or large instead of whether there were one or two. The Stroop task was much the same as that used in (Chein & Morrison, 2010), except neutral trials were used in place of congruent trials (non-color text was used with each font color).

Results of this experiment are shown in blue in Figure 5.8. Switching costs in Figure 5.8a refers to the difference between the average amount of time to do a trial when switching a task and the average amount of time to a trial that is a repeated task. Training significantly transferred to all these test tasks. Interestingly, in the Stroop test, the control case of training only on single-task blocks worsened WM interference.

The Actransfer model of this experiment again bases transfer on the decision to prepare during inter-trial wait periods. During training, the agent always practices the prepare operation between trials, while during testing it has the choice to either prepare or be idle. Practicing the prepare operation biases the agent to prepare during testing. Practicing the single-task case in the control setting, by contrast, forces the agent to practice not preparing, which biases the agent to not prepare

[3]Actransfer was also used to model a WM span task from the original human experiment. I ignore this model because it primarily demonstrates ACT-R's declarative learning mechanism, which is already well established, rather than procedural learning with PRIMs.



(a) Task switching test

(b) WM digit span test

Figure 5.8: Human data from Karbach and Kray (2009) and Actransfer model data from Taatgen (2013) in the task-switching transfer experiment.

during subsequent Stroop testing.

Original Actransfer model results are shown in dark red in Figure 5.8, and Actransfer results omitting $T_{retrieve}$ in bright red. Model results are the average of 21 experiment repetitions. Again, the original model captures the same transfer trends, though fits are inexact. The Actransfer `(no RT)` model is closer to human performance in the task-switching test, because in this case $T_{retrieve}$ does not add to its switching costs. In the Stroop test, however, `(no RT)` once again leads to negative interference.

CHAPTER 6

Soar

The design of PROPs draws deeply from the underlying theory and design of Soar to unify the architecture with PRIMs theory. I must therefore explain the key principles of Soar that influence the PROPs design.

I do not attempt to cover all aspects of the architecture, but only those necessary for understanding the contributions of PROPs. For a more detailed discussion of Soar's components, see appendix A.

6.1 Operators

Soar defines procedures and decision making differently than ACT-R. In the ACT-R tradition, a single decision corresponds to the selection and firing of a single *if-then* rule in procedural memory. Only one rule fires per decision cycle. In Soar, a decision cycle corresponds to the selection and application of a single operator. An **operator** is a special structure in the architecture that is created, selected, and applied by multiple rules to guide decision making. In Soar, whenever a rule can fire, it will fire, in parallel in practice with all other matching rules.

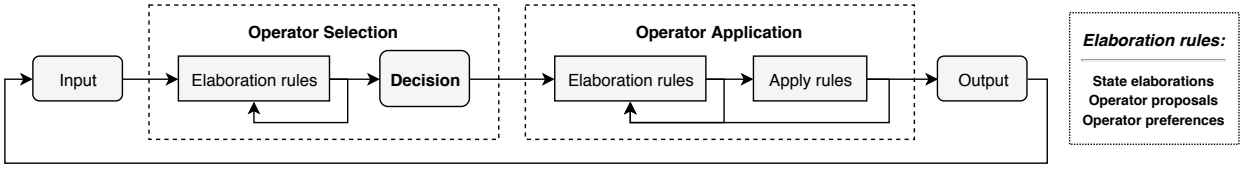


Figure 6.1: The Soar decision cycle, adapted from (Laird, 2012).

Figure 6.1 shows the Soar decision cycle, which is divided into the operator selection stage, and the operator application stage. Operator selection begins after the Soar agent receives input from its environment. In this stage, the agent selects a single operator as its decision for that cycle. In the operator application stage, the agent then performs various actions that carry out that decision.

There are two main types of rules that can fire during these stages: elaboration rules, and apply rules. **Elaboration rules** fire during both stages. **Apply rules** only fire during the application stage. Elaboration rules come in three kinds, *state elaborations, operator proposals,* and *operator preferences.* In both stages of the decision cycle, any matching elaboration rules fire first, and if the effects of these allow more elaboration rules to fire, those will also fire before proceeding to the next part of the cycle. The WM changes made by an elaboration rule only persist so long as the rule's conditions match, but changes made by an apply rule persist until changed by additional rules. Since apply rules only fire in response to selected operators, operator decision making is what causes persistent changes to WM.

An **operator proposal** elaboration rule makes an operator available for the architecture to select in decision making. A proposal rule's actions create a special "operator proposal" declarative structure in WM that tells the architecture about the potential decision, and which remains in WM for as long as the proposal's conditions are satisfied. As elaboration rules, proposal rules can fire during any stage of the decision cycle, but the operator proposals they create only affect the decision made at the end of the selection stage. If there are multiple proposed operators, **operator preference** rules can create preferences to select among them. Preference rules normally use symbolic logic, such as "IF (`get-command` and `idle` are both proposed) THEN (prefer `get-command` over `idle`)." Soar also supports Reinforcement Learning (RL) preference rules, which add or subtract utility to a proposed operator within the architecture, so that Soar can use built-in probabilistic selection policies to select among operators based on those utilities. Any other elaboration rules that do not create proposals or preferences are simply called **state elaboration** rules. These can be used to support other information in WM that could influence decision making. Once no more elaborations fire in response to input, the architecture selects the preferred operator as the decision for that cycle. Then the operator application stage begins, in which apply rules fire to carry out the chosen operator. Additional elaboration rules can also fire in response to apply rules. After no more rules fire in the operator application stage, any output is sent to the agent's environment, and the cycle repeats.

This contrasts with the ACT-R/Actransfer design where each rule is available for decision making based on that single rule's conditions, where the decision is made based on the utility values of the matched rules, and where the decision is applied according to the selected rule's actions. Soar uses separate rules for each of these three processes in a decision cycle.

6.2 Working Memory

When a rule fires, it modifies Soar's WM. Figure 6.2 depicts a sample portion of Soar's directed cyclic graph WM. Each node (circle) represents a WM *identifier* (ID), and each edge (arrow)

represents an *attribute*. Nodes can have any number of incoming or outgoing edges. Each edge points from an ID to a single *value*, either another ID or a constant number or string. Each edge corresponds to what is called a Working Memory Element (WME), defined by an id:attribute:value triple. The WM graph is always rooted in a *state* ID, `S1` in the figure, and each possible edge path through the graph from that state ID defines a unique *WME address*, a path for accessing that WME from the root. Thus, Soar's graph WM is not a fixed set of buffers or slots, but is unbounded in size and structure, able to change during agent processing as memory operations add or remove WMEs.

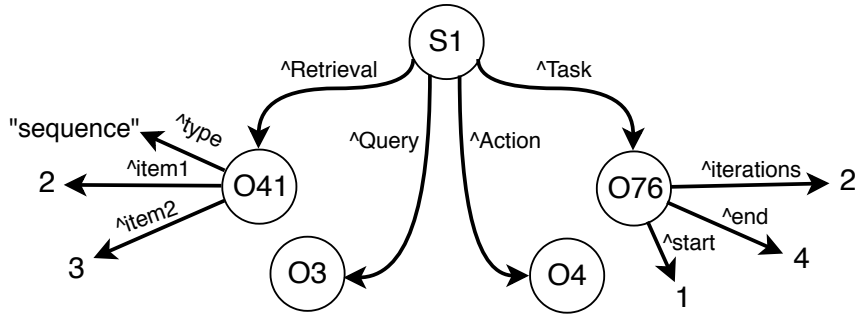


Figure 6.2: Example Soar WM structure. WM is a directed graph rooted in a single state ID (shown as S1). Graph edge names are preceded with ∧.

Soar rules use variables to match the contents and structure of the WM graph. As a pseudo-code example, using the structure in Figure 6.2, a rule could test a condition `(S1.Task.<var> > 2)`. The variable `<var>` would bind to `end`, since the value at `S1.Task.end` is greater than two. Similarly, a rule might test the condition `(S1.<var1>.<var2> == 2)`, and this general condition is satisfied if any value that is two edges away from the root had a value of two. In this case, the condition matches twice, on both `S1.Retrieval.item1` and `S1.Task.iterations`.

6.3 Long-term Declarative Memory

Like ACT-R, Soar has LTDM storage for symbolic memory content. In Soar, LTDM is divided into two long-term memory systems, **Semantic Memory (SMEM)**, and **Episodic Memory**. SMEM can hold any number of graph-based structures, either stored deliberately by the agent from its WM contents or stored by the agent designer before the agent's operation begins. Episodic Memory automatically records snapshot episodes of the agent's WM content as it runs. I use SMEM as the equivalent of ACT-R's LTDM for the purposes of this thesis (namely as the storage system for task instructions), and I do not discuss Episodic Memory further. I use the term LTDM when referring to the long-term declarative memory system prescribed by PRIMs theory as a whole or

to Actransfer's specific design, and I use SMEM when referring specifically to Soar's long-term declarative memory system.

6.4 Problem-Space Computational Model

Soar decision making is organized according to the Problem-Space Computational Model (PSCM) (Newell, 1990). Newell and Simon (1972) define a problem space as having three components: 1) an *initial state* with a goal that needs to be achieved, 2) a set of *operators* relevant to that goal that can be used to change WM to achieve it, and 3) the ability to *test* whether a particular state achieves the goal. The purpose of the Soar decision cycle is to select and apply operators to progress the WM state toward the goal state.

In Soar, if an agent does not have operator proposal, preference, or apply rules that let it progress in its problem space, this is called an *impasse.* An impasse will also occur when a selected operator is too complex to apply with a single decision cycle. An impasse needs to be resolved in a child problem space before decision making can continue in the original parent problem space. When there are impasses, the Soar architecture maintains a stack of WM states, each associated with a particular problem space, and each a graph with its own root state ID. Each state's problem space has its own set of relevant operators that can be proposed and which modify that state toward achieving its particular goal. A state is added to the stack when an impasse arises. The implicit goal in a newly created child state is to resolve the impasse of the parent state. Once agent processing solves an impasse, such as when substate operators apply a parent state's operator that had been missing an apply rule, then the parent state operators can continue for the first problem space, and the child state is removed from the WM stack. In Soar terminology, the stack of states is also referred to as a stack of *goals* and *subgoals*.

Figure 6.3 depicts a stack of Soar goals, with one main task goal and two nested subgoals. The inner rectangle on the left shows the space of operators proposed for the first task goal. In the middle is a goal to resolve an impasse in the first problem space, in this case the problem of applying the operator selected in that first problem space. On the right is a problem space for applying an operator selected from that middle problem space.[1]

Soar determines whether a particular WM (sub)state achieves a goal by testing whether the agent can continue its decision making for the parent goal (by selecting and applying operators). In the sub-subgoal frame on the right in the figure, the selected operator does not cause another subgoal because the agent has rules that can apply it. If that operator changes the parent WM state such that its impasse is resolved and decision making is able to continue, it continues there, and the

[1]There are other ways the PSCM and substates can be used beyond applying parent-goal operators. I do not describe these for brevity.

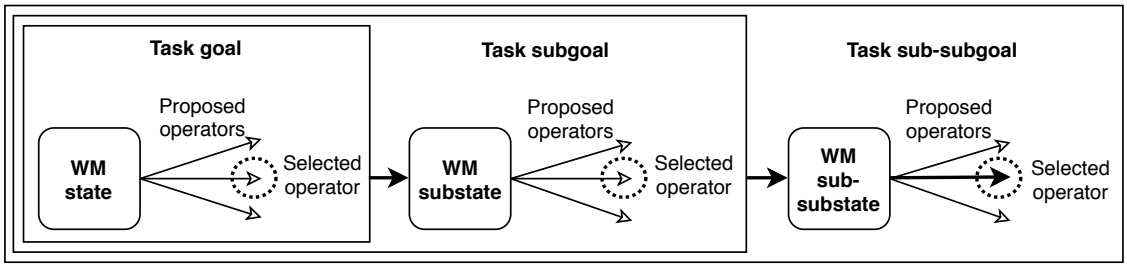


Figure 6.3: The Soar PSCM model for hierarchical operator execution, adapted from (Laird, 2012).

child state is removed from the stack. Because states are added for impasses in decision making, the decision cycle of operator proposal, selection, and application happens only for the deepest state in the stack. Impasses can also be resolved when environment perception changes to allow decision making to continue in a parent problem space. For example, if an agent is busy typing text into a computer, and someone asks it to stop and switch to a card-sorting task, this makes the original goal and its impasses irrelevant. In this case, Soar then removes any irrelevant descendant subgoal problem spaces and their associated WM state structures from the stack and continues decision making in the main task problem space (for the different task).

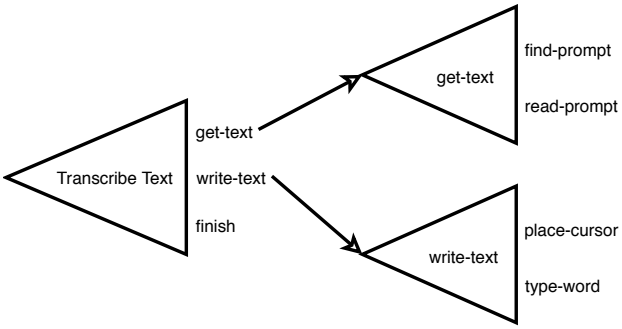


Figure 6.4: Hierarchical problem spaces for the transcribe-text task.

In the PSCM, an agent's task behavior can be described in terms of a hierarchy of goals and problem spaces. Figure 6.4 shows a hierarchy of problem spaces for the transcribe-text task. In the figure, each problem space is shown as a triangle, with operators relevant to that problem space shown to its right. The goal for the main "Transcribe Text" problem space is to copy a prompted line of text into a computer text editor. The problem space involves two main operators: get the text from the prompt, and then write the text into the editor. Getting the text requires finding the prompt and reading the text, as shown with the child "get-text" problem space. The "write-text" problem space for writing the text requires placing the edit cursor at the correct location before typing the prompted text. The finish operator is shown to not require its own problem space or subgoal. It can be carried out with a single apply rule. This arrangement of problem spaces for this task is not the only possible arrangement. One can imagine other valid hierarchies that also

achieve the task goal, including one where all operators are included in one problem space. One can also imagine a hierarchy where there are even more problem spaces for other operations such as checking how far along the writing process is or checking for whether one needs to stop and do some other task like card-sorting or for directing more fine-grained operations such as typing individual keys. There is no theoretical limit to the depth or branching factor of a problem space hierarchy.

6.5 Chunking

"Chunking" is Soar's procedural learning mechanism. It creates rules that summarize subgoal problem solving so that it can be quickly replicated in the future without using a subgoal. These learned rules are called *chunks*.[2]

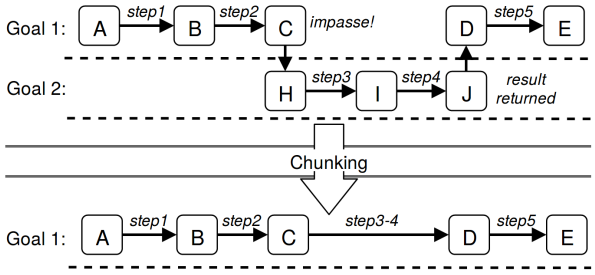


Figure 6.5: Soar chunking.

Figure 6.5 depicts the chunking process. Boxed letters represent a WM state at different points in time, and arrows represent operators that modify WM. Decision making for "Goal 1" requires a subgoal to resolve the impasse at state C. The architecture creates a new WM substate for the new subgoal, labeled "Goal 2." Two subgoal operators then perform the steps needed to resolve the impasse in C, and these require two decision cycle steps, 3 and 4. After step 4, a rule modifies WM in state C, resulting in D, from which decision making can continue. Soar chunking summarizes the final result of these subgoal operators into a single learned rule, such that the next time the agent is in the same scenario as C, it applies the result immediately with the chunk in one decision cycle.

The computation of Soar operators, WM, and the PSCM with chunking has broad implications for the PROPs system implementation, and gives rise to its name as the *PRimitive OPerators* (PROPs) system.

[2]Unrelated to ACT-R declarative *chunks*.

CHAPTER 7

PROPs Iteration 1: Defining Support for Working Memory Operations

In this chapter I describe $PROP_1$, the first iteration of the PROPs system. As I described in my methodology in chapter 3, the purpose of $PROP_1$ was to replicate Actransfer's computation as much as practically possible so that in future iterations I could attempt to address gaps in the model. Any substantial computational differences between Actransfer and $PROP_1$ should only be because some aspect of the Actransfer computation is not readily compatible with Soar.

Actransfer's implementation was indeed incompatible with a Soar implementation in several ways. The chief issue was reconciling Actransfer's definition of primitives, P1 in the PRIMs flow diagram, with Soar's definition of WM. Soar's WM does not have a fixed size and structure that allows me to define the initial set of primitives in the way that was done for Actransfer. Thus, in order to replicate Actransfer I also flesh out the computational definition of P1 and the theory of how PRIMs relate to WM. This is circled in blue in Figure 7.1, which otherwise is the same as the flow diagram in Figure 4.1.

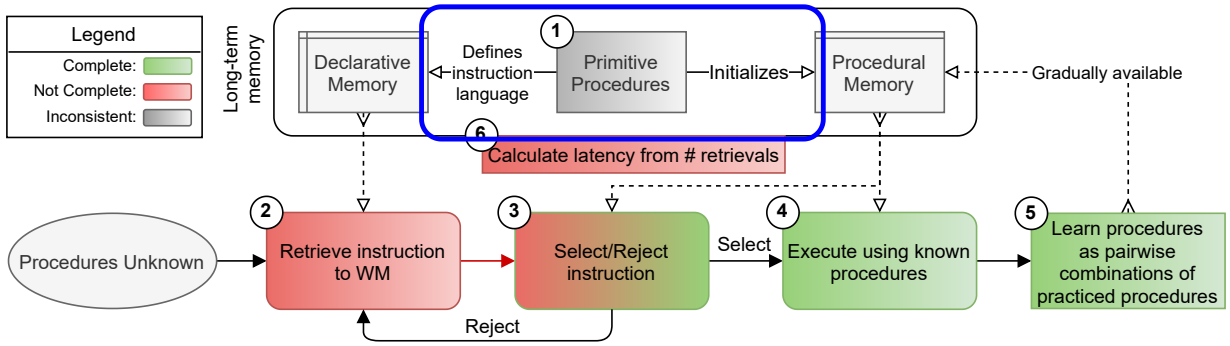


Figure 7.1: A review of the Actransfer flow diagram. $PROP_1$ addresses incomplete computation circled in blue.

The result of this work led to an extension of PRIMs theory. I introduce a new primitive process that explains where the PRIM rules of Actransfer come from and how an agent can *learn*

PRIM rules for any specific WM locations starting with only innate procedural knowledge for each generic PRIM operation *type*.

While designing PROP_1, I identified other smaller inconsistencies between Actransfer's approach and Soar's. In the next section of this chapter, after I describe how I address the P1 problem, I describe each of these and our solutions. Then I present and discuss the evaluation of PROP_1.

7.1 Introducing A New Primitive Operation

Actransfer's P1 that defines PRIMs from permutations of WM slots is incompatible with Soar's theory for WM structure. Soar represents WM as an unbounded graph, which can grow or shrink or change in structure as edges and nodes are added or removed throughout the lifetime of the agent. Thus, there is no fixed set of WM addresses or slots. Rather, Soar's unbounded graph would entail infinitely many unique PRIMs for all the ways of adding or removing WMEs to form different graphs. Even if Soar did not have an unbounded WM space, the combinatorics of PRIMs are already substantial. Even a small number of WM slots requires an enormous number of PRIM rules for all possible operation combinations among them.

I could have replicated Actransfer by constraining Soar WM to a graph that is isomorphic to the configuration of slots Taatgen (2013) used for Actransfer. The root node in the graph could point to nodes labeled "const1," "action1," "slot1," and so on for every Actransfer slot. Then I could initialize procedural memory with rules specially designed to use each specific slot-like node in the graph. However, I observed that I could instead introduce a true solution for P1 that 1) is consistent with Soar theory for WM, and 2) which I can enable or disable in such a way that, when it is disabled, the computation is equivalent to that of Actransfer. This accelerates PROPs evaluation and lets me establish the first iteration PROPs system in a manner that is more consistent with Soar.

I address the P1 problem by introducing a new type of primitive operation that supports the original PRIM operations of Actransfer. In Soar, I represent this as a new subgoal and problem space whose operators construct the behavior of a specific PRIM operation for specific WM elements, according to whatever is instructed by condition or action lines. When chunking summarizes the results of this subgoal processing, the agent *learns* chunks that carry out the condition and action lines of its task instruction, equivalent to the PRIM rules that Actransfer begins with. I am able to toggle this learning on or off by initially providing the agent with the chunks that it would otherwise learn through this process.

The PROP_1 agent begins with only a *single* rule instance in procedural memory for each PRIM operation type, not a different instance for each possible use of that type across WM. That is, the agent starts with a single generic `(<x> == <y>)` rule for all equality conditions, a single `(COPY`

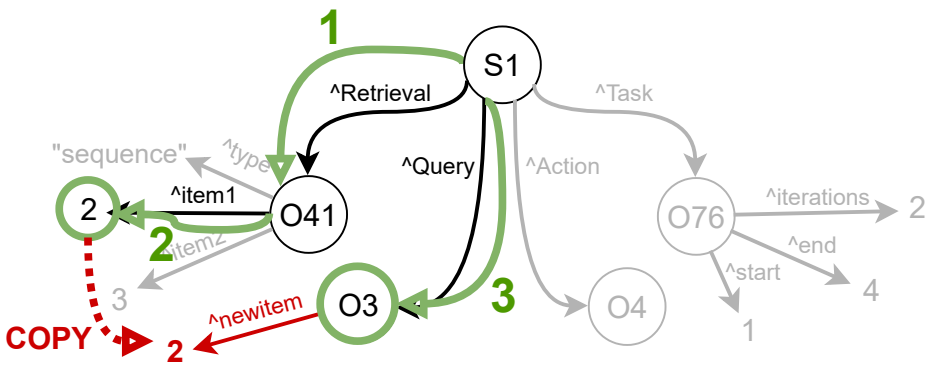


Figure 7.2: PRIM resolution steps in Soar WM for a COPY operation.

`<x> to <y>)` rule for all copy operations, and so on. The goal of the new subgoal processing is to bind the variables (`<x>, <y>`, etc.) for a single generic rule to specific WM locations. I call this subgoal processing *PRIM resolution.* The subgoal processing of PRIM resolution "resolves" the unbound generic PRIM-type rule by binding it to specific locations in the WM graph.

Figure 7.2 demonstrates the subgoal processing of PRIM resolution in more detail. This figure specifically shows PRIM resolution for the action, "COPY Retrieval.item1 to Query.newitem." This action should create a new edge in the agent's WM graph at "Query.newitem" with the value found at "Retrieval.item1." But the agent begins with only a rule for `(COPY <x> to <y>)`. Because the agent lacks a specific rule for this action, this creates an impasse, and the agent enters a PRIM resolution subgoal. In this subgoal, the agent first attempts to resolve the variable `<x>`. It reads the action line and finds the text, "Retrieval.item1." This describes two connected edges in the graph. Subgoal operators interpret this label and trace the corresponding path through WM to find the referenced memory element. This requires two steps, one for each traversed graph edge, as shown in the figure with green arrows "1" and "2." Once the agent has traced this path and found the referenced element (in this case, the number circled in green), it can return this element as the result of subgoal processing and also learn a chunk that can repeat this process in the future for this WM reference. At this point, the agent is still missing the binding for variable `<y>`, and so the agent enters another PRIM resolution subgoal. In this subgoal, it similarly reads the action line to trace the single graph edge "Query" to find the reference `O3`. The last edge in the action line, "newitem," will be created, so PRIM resolution stops the trace at `O3`. It returns this reference to the parent state, and the PRIM COPY operation is now resolved to `(COPY Retrieval.item1 to Query.newitem)`. The net effect of PRIM resolution is thus to convert a declarative description of a WM element into a usable procedural reference.

This slightly modifies the computational meaning of a PRIM. In my PROPs system, PRIM rules are no longer operations that are bound to specific WM slots, but rather they define generic

operation *types* (such as add, remove, etc.) that can apply to any WM location.

Note that chunks learned from the PRIM resolution process are transferable wherever the same variable reference is used for a single instruction line argument. This is a finer level of transfer than even the lowest level of PRIM combinations in Actransfer, which only transfer when the same WM slots are used in a pair of two different PRIM operations. While a PROPs agent using PRIM resolution initially requires more processing than an Actransfer agent, it quickly learns all the chunks it needs to perform as quickly as Actransfer for its given task instructions. The PROPs agent using PRIM resolution also has a provably smaller procedural memory footprint than an Actransfer agent. The proof for this is trivial. Actransfer requires procedural memory to begin with a PRIM rule for every *possible* permutation of every primitive operation in WM. A PROPs agent only learns chunks for operations that task instructions require in practice, which are less than or equal to the full set of Actransfer PRIM rules. Further, the PROPs agent only needs a single chunk per single variable binding, rather than a rule for every *pair* of variable bindings. This further reduces the footprint. In practice, the PROP_1 agent only needs to learn about 100 PRIM resolution chunks for a single task in the Actransfer experiment suite, in contrast to the 1,693 PRIM rules that Actransfer must start with when using 31 WM slots.

In summary, I introduce a new layer of primitive procedural memory processing and transfer to the PRIMs theory of procedural learning. As I describe later in section 7.7, this extra layer of primitive processing leads to a more human-like power-law learning and transfer profile that is absent from Actransfer results (Stearns et al., 2017). PRIM resolution could theoretically be applied in other production system architectures besides Soar, potentially in ACT-R, even if they do not have Soar's unbounded or hierarchical WM.

7.2 Distinguishing Primitive Operators from PRIMs

A PRIM rule in the original sense defined by Taatgen (2013) is both a primitive *rule* and a primitive *decision cycle operation*. This follows from the way that a single fired rule in ACT-R represents a single decision cycle. This is not the case in Soar, where there is a single *operator* per decision cycle but potentially many proposal, preference, and apply rules fire in parallel. And yet the agent does not begin with innate operators but innate rules. Operators do not exist in a Soar agent's procedural memory as known rules, but rather they are constructs created on-line by proposal rules and applied by apply rules. Thus, a primitive rule in Soar is not the same as a primitive decision cycle operation (a rule in ACT-R).

I introduce a new construct to PRIMs theory called the *PRimitive OPerator* (PROP), for which the PROPs system is named. From here on, I refer to the innate, primitive rules in the PROPs agent as "PRIM rules" or "PRIMs," in the same sense as used in Actransfer, and to the primitive *decision*

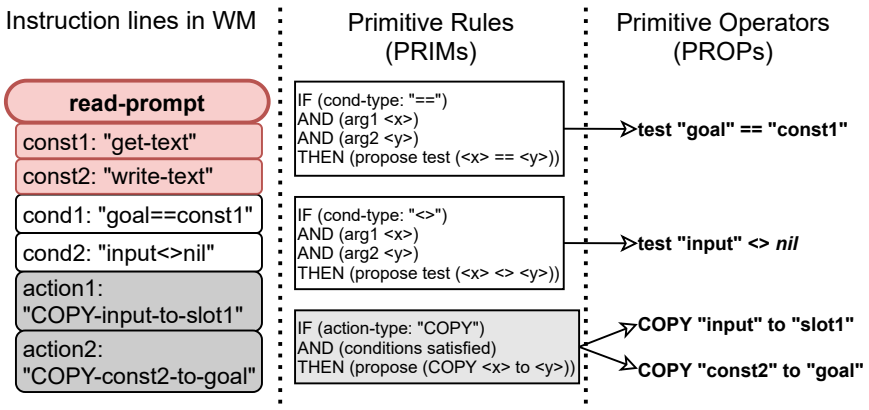


Figure 7.3: PRIMs and PROPs in the PROPs system for the "read-prompt" instruction.

cycle operations done by the agent as "PROPs." In the PROPs system, the general, innate rules such as `(COPY <x> to <y>)` that can be bound to specific WM elements are the PRIM rules. When the PROP_1 agent retrieves a task instruction into its WM, it can use PRIM rules to propose a PROP specific to each instruction line.

Figure 7.3 depicts the distinction between PRIMs and PROPs in the scenario when an agent retrieves the "read-prompt" instruction into WM. The agent has three relevant PRIM rules that can match on this instruction. (Not shown is the PRIM resolution process that binds the instruction line arguments to the PRIM rule variables.) Each of these proposes a PROP that carries out the instructed primitive operation. (Not shown are other PRIM rules that would apply these operators if they were selected.)

There are four main categories of PRIM rules in the PROPs system. The first is for primitive condition lines, the second is for primitive action lines, the third is for operator preference elaborations, and the fourth is for supporting the overall process of proposing operators from instruction lines. Table 7.1 lists the PRIM rules types for the PROPs system according to these four categories. The number in parentheses after each entry in the first three columns is the number of WM location arguments required for PRIM resolution to resolve a PRIM. In PROP_1 there are two PRIM rules for each of these types, one to propose the corresponding PROP and one to apply it.

Conceptually, the first two categories for conditions and actions include the same rule types as those that exist in Actransfer, although the Soar architecture supports slightly different operations. For instance, Soar tests whether a WM graph element does not exist rather than whether it has a *nil* value, and it can add an ID node to the graph as an action. The third category of Soar PRIMs, operator preferences, allow task instructions to guide operator preference. Actransfer/ACT-R does not use preference rules, so this category is irrelevant to replicating Actransfer. I show it here for completeness, since preference rules provide a class of primitive memory operation in Soar.

The fourth category, proposal support, is unlike the others. The first type of PRIM rule in this

Conditions	Actions	Preferences	Proposal Support
Equal (2)	Copy (2)	Acceptable (1)	PRIM resolution
Unequal (2)	Remove (1)	Indifferent (1)	Propose actions
Exists (1)	Add ID (1)	Better (2)	
Not Exists (1)		Worse (2)	
Type Equal (2)		Best (1)	
Greater (2)		Worst (1)	
Greater/Equal (2)		Reject (1)	
Less (2)		Require (1)	
Less/Equal (2)		Prohibit (1)	

Table 7.1: Primitive memory operation types through which PROPs are proposed and applied.

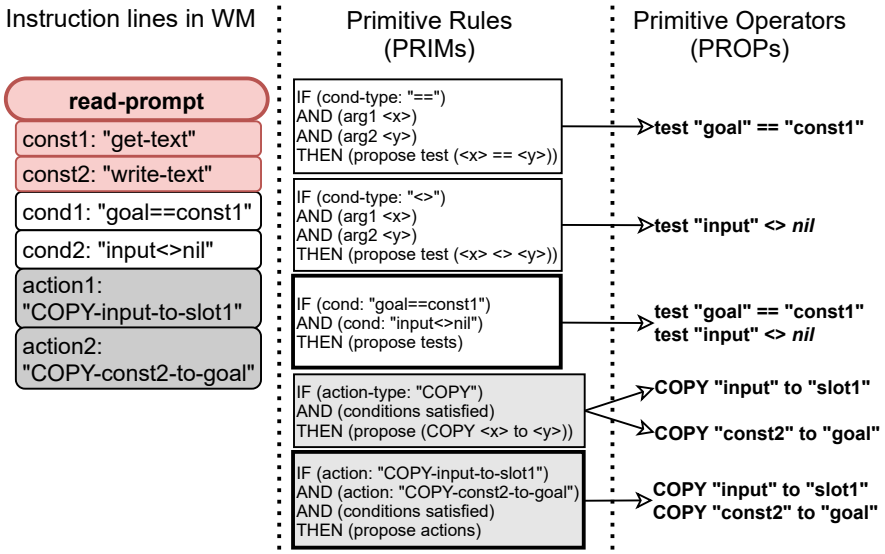


Figure 7.4: PRIMs and PROPs in the PROPs system for the "read-prompt" instruction after the agent has learned compositions of PROPs.

category provides PRIM resolution behavior. The implementation of the PROPs system in fact uses multiple rules to perform PRIM resolution, but the net behavior is a single process, and so it is listed as a single entry in Table 7.1. The second type of PRIM rule in this category is the rule that allows the agent to propose instructed action lines after all the associated condition lines are found to be satisfied. This is a single PRIM rule that fires when the count of satisfied condition lines is equal to the total number of condition lines.

My work with PROP_1 showed that this design allows the PROPs agent to generate PROPs for general memory operations on-line for any structure of Soar WM (Stearns et al., 2017).

When learning combinations of WM operations (P5), PROP_1 learns to compose PROPs by learning chunks (rules) for pairs of *proposal* rules as well as for pairs of *apply* rules. Initially, given task instructions in WM, the agent proposes an operator for each primitive operation, as

shown in Figure 7.3. With chunking, the agent learns proposal rules for operators that apply multiple primitive operations at once, and in order to apply these operators the agent must also learn the apply rules that carry out multiple operations at once. Unlike PRIM rules, these chunks are already bound to specific WM elements. Figure 7.4 shows the PRIM rules from Figure 7.3 alongside chunks for composed PROPs, drawn with bold lines. (The PROP_1 agent prefers to use these larger compositions of PROPs.)

In terms of cognitive theory, PRIM rules are supposed to represent primitive procedural knowledge. PROPs, by contrast, represent primitive processing done using procedural knowledge. The distinction that I introduce here is that there need not be a one-to-one mapping between these. A small set of primitive procedural knowledge can support a much larger set of primitive memory operations.

7.3 Supporting Ordered Retrievals

As described earlier, P2, the retrieval selection phase, presents a significant gap in Actransfer's implementation of PRIMs theory. Actransfer does not define how the agent knows which instructions are appropriate to retrieve from LTDM. Actransfer includes a custom, automatic architectural process that tests all conditions of all instructions and then increases the activation of instructions that have satisfied conditions. This way, the agent retrieves an instruction that is practically already guaranteed to have satisfied conditions when it tests them in P3. Clearly, this custom architectural processing is performing the test work of P3 ahead of time so that the agent does not use many decision cycles in P3 on instructions that it has to discard. This is not part of PRIMs theory, but it at least serves to let Actransfer run in absence of an more detailed model for P2-P3.

My goal for PROPs is to fill such gaps. However, as described in chapter 3, for PROP_1, I first implement Actransfer functionality in Soar as closely as possible so that I can evaluate iterative changes that I then introduce. But, as also described in chapter 3, I do not wish to do this by modifying the Soar architecture. Therefore, I do not attempt to replicate the custom architectural modification of Actransfer that bias declarative retrievals toward instructions with satisfied conditions.

Given that the main effect of the Actransfer P2 approach is to make it likely that the agent never retrieves an instruction it cannot execute, I instead have the PROP_1 agent deliberately reason over rote declarative knowledge of the sequence of instructions it needs to retrieve (e.g. "$Instr1 \rightarrow Instr45 \rightarrow Instr2 \rightarrow Done$"). The agent keeps an explicit declarative, task-specific representation of a sequence of instructions in its long-term SMEM storage. This is the first thing the agent retrieves into WM when given the name of a task to perform. If the agent retrieves (and applies) instructions according to this sequence, the instructions it retrieves always have satisfied

conditions.

This assumes that the task allows such deterministic behavior, and this assumption is not valid for all experiments in the Actransfer suite, particularly the WM/Stroop and task-switching experiments. However, it is sufficient for an initial replication and test of PROP_1 when deterministic behavior is possible.

The PROP_1 agent always retrieves instructions with satisfied conditions, so long as the given sequence is reliable. Since Actransfer uses random noise to moderate its retrievals, and can therefore occasionally retrieve an unsatisfied instruction, PROP_1 has slightly fewer rejected retrievals. Since retrievals take time, PROP_1 ought to therefore be slightly faster in task performance. However, as is discussed in section 7.7, I show that the net behavior is comparable.

This design leaves the same gap for P2 that Actransfer did and should come close to replicating its behavior without the need to modify the architecture. It could, however, also be a simple model of a human task subject who follows a memorized routine of instructions or who remembers an instructor's demonstration of each step for a task.

7.4 Simulating Gradual Learning

As described earlier, Actransfer uses ACT-R's procedure compilation mechanism to implement gradual learning (P5). Any time the agent fires two rules in two sequential decision cycles, the architecture combines the two rules into a new learned rule, if possible. This learning is one-shot, but the learned rule starts with a low utility, which means that at first it is not likely to be used. But its utility increases each time the agent repeats those two component rules in sequential decision cycles the same way it did to generate the learned rule. Eventually, the utility of that learned rule increases enough so that it is used in place of its component rules. The net effect of this approach is that the agent must practice using its primitive rules for several repetitions before it actually uses learned rules in their place.

Since gradual learning is a key element of PRIMs theory, I must include this behavior in PROPs. However, while Soar chunking is also one-shot learning, Soar does not define any gradual process for adopting learned rules the way ACT-R does. I therefore do not have the option of using chunking directly to emulate the Actransfer gradual learning process, and I do not wish to modify the architecture.

PROP_1 simulates gradual chunking by using agent decisions to mediate and control the learning process, such that the agent deliberately makes its learning gradual. I designed the PROP_1 agent to keep declarative knowledge of all pairs of component rules that it practices together and to track how many times each pair has been seen together during practice. If this co-occurrence count passes a parameterized threshold, which I here call θ_p, then and only then does the agent turn on

chunking for that substate and learn the rule. This simulates ACT-R gradual learning by delaying how long it takes to add a rule to procedural memory rather than by delaying how long it takes to use that rule after it is added. This difference is necessary because Soar will always fire a chunk whenever its conditions are satisfied. Setting θ_p to 1 is equivalent to standard one-shot chunking in Soar. If θ_p was set to 2, then the architecture must attempt to create a chunk twice before it actually stores it in procedural memory, and so on.

As explained in my methodology in chapter 3, during experimentation I omit agent task latency that is due to agent decisions such as these that replicate an Actransfer architectural process that has not yet been implemented in Soar. These decision cycles fill in for the required processing until future PROPS iterations can address the problem directly.

In summary, I identify how Soar theory can be expanded to support gradual procedural learning in a manner consistent with existing architecture theory. As I will discuss in section 7.7, experimentation showed that this approach to gradual procedural learning can provide equivalent or even superior learning profiles compared to the gradual utility-based learning of ACT-R for these tasks (Stearns et al., 2017).

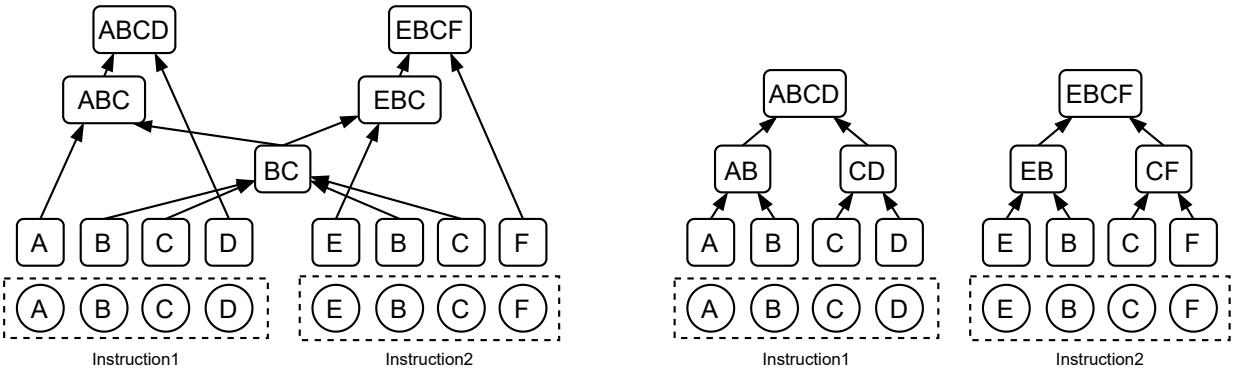
7.5 Computational Motivations for Gradual Learning

In this section, I briefly pause my description of PROP_1 in order to explain a theoretical principle I uncovered in my research that is crucial to understand how and why I used Soar to learn primitives the way I did. This is the principle of *why* an agent should learn rules gradually. PRIMs theory requires gradual learning as a means for emulating human learning patterns. But what are the computational effects of gradual learning, and does it serve a practical benefit? If a rule could be learned and used now rather than later, why not learn it immediately and reap the performance benefits right away?

I found that gradualness is essential for PRIMs theory learning to ensure proper *transfer* of the learned rules. In broader psychology research, it is known that the variability and rate of the skills practiced during training can shape transfer (National Research Council, 1994). However, while there continue to be many studies into how gradual skill *practice* can shape human transfer (Sabah et al., 2019; Sawers & Hahn, 2013; Shahar & Meiran, 2015), I am unaware of any prior work in the cognitive architecture field that has discussed the computational motivation for why the learning mechanism should compose rules gradually to enhance transfer.

As Stearns et al. (2017) observed, gradual learning in PRIMs theory serves the function of letting the architecture experience which PRIM combinations are used the most across multiple practiced instructions before favoring a particular learning path. Consider the example in Figure 7.5, in which there are two instructions, "Instruction1" composed of primitives A, B, C, D and

"Instruction2" composed of E, B, C, and F. If the agent waits to compile the first layer of new rules until after practicing both Instruction1 and Instruction2, experience reinforces to the architecture that B and C appear together more frequently than other primitives across both instructions. Thus, the architecture prefers the learning path shown on the left in the figure, and the agent benefits from transfer. However, if the agent instead learns greedily with one-shot learning, it might perform Instruction1, be given the option of composing either (AB), (BC), or (CD), and randomly choose (AB). Then it would be forced to follow the learning path shown on the right, which allows no transfer across these instructions.



(a) Transfer

(b) No Transfer

Figure 7.5: Examples of different rule learning paths over time, based on which pairs of rules are compiled first. Circles indicate instruction lines; rectangles indicate rules that execute those lines. Learning path (a) allows transfer between the instructions. Learning path (b) does not.

Anderson (1982) similarly observed that an agent that uses pairwise production compilation can risk learning new rules that are useless or which can even prevent the agent from learning more useful rules. His proposed solution, however, was not gradual learning but to use subgoaling to group productions that could more safely be composed.

Some theoretical work in researching the neural mechanisms of human rapid instructed task learning predicts that a faster learning rate in the Pre-Frontal Cortex (PFC) regions that build representations for task skills might actually cause slower overall task learning (Cole et al., 2013). Their argument is that a higher learning rate would result in overfitting, and that this would therefore result in less-generalizable compositional structures, which would slow down progress in the long-term. This is essentially the same concept I describe here and experimentally demonstrate later in this chapter in section 7.7.3.

7.6 Combining PRIMs as Sets

Actransfer uses ACT-R's production compilation mechanism to learn rules, and this entails a particular approach to building rule combinations that is not directly compatible with Soar theory. As described in chapter 4, the ACT-R mechanism compiles rules that are sequentially adjacent to each other during task practice. If an Actransfer agent practices A->B->C->D, in that order, it can learn the combinations (AB), (BC), and (CD), but not, for instance, (BD). But in theory the relative order of execution does not matter for primitive conditions or actions within an instruction, so long as the conditions are tested before the actions are executed. If A, B, C, and D are all conditions or all actions, (AB), (AC), (AD), (BC), and (BD) should all be valid combinations. That is, PRIMs theory regards conditions and actions as *sets* rather than *sequences.*

This presents a gap in Actransfer's implementation that I desired to address with PROPs. This was particularly relevant for me to address with $PROP_1$ because the design of Soar chunking, which is based on summarizing substate processing, does not natively support the sequential compilation approach of ACT-R. I could not easily implement Actransfer's sequence-based learning approach. But Soar could easily support a set-based learning approach, similar to that which is actually desired for PRIMs theory.

There are multiple ways that I could have made $PROP_1$ combine primitives as sets. The first approach I investigated was to have the agent compile all possible pairs of practiced conditions or actions during each iteration of practice. For instance, if the agent practiced A->B->C->D, in that order, it could learn the combinations (AB), (BC), and (CD) like Actransfer, but also learn (AC), (AD), and (BD) from that same round of practice. However, this approach highlights a combinatorics problem. The learned rules can then be practiced alongside more primitive rules and thus can be combined with them in future iterations of learning. This means that each newly learned rule could lead to many more future rule combinations. For instance, after learning (AB), (BC), (CD), (AC), (AD), and (BD), the next layer of learning could include ((AB)(CD)), ((AB)C), ((AB)D), (A(BC)), ((BC)D), (A(CD)), (B(CD)), ((AC)(BD)), ((AC)B), ((AC)D), ((AD)B), ((AD)C), (A(BD)), and/or ((BD)C), fourteen more rules, many of which are redundant. Continuing this hierarchy of learning could in the worst case lead to a total of 32 rules learned for the sake of the four primitive operations. This number would grow significantly for each additional primitive added to the set in a single instruction. And an agent can have many instructions for a single task, each with its own set of operations.

Brief experimentation showed that this kind of learning significantly impairs the agent's ability to run. Running Soar on my lab desktop computer, the agent would grind to a near-halt after only a few cycles of learning. In this case, the slowdown came from the architectural process of *matching* the learned rules. As the number of known rules for applying a sequence of primitives grows with

learning, so also does the workload for the architecture to update which rules match and which do not. In this case, the growing number of *proposal* rules particularly exacerbates the problem. When there are multiple competing operator proposals, the agent also needs to use *preference* rules that match on pairs of proposals to give relative preference among them so that the agent can choose one. This multiplies the processing workload by a factor of up to $\binom{n}{2}$, where n is the number of proposals, and a preference rule matches on 2 proposals. Thus, for each proposal rule the agent learns there is a combinatorial cost to compete it against other proposal rules. But the more preference rules I use to filter out proposals, the greater the immediate processing load to match the preference rules. However, the fewer preference rules I use, the less I can control whether the agent even uses newer learned operator combinations rather than more primitive operators, and this would decrease the effectiveness of learning new rules. Clearly there needs to be a more principled constraint on rule learning.

My goal for PROP_1 was to find an approach that allows conditions and actions to be treated as sets rather than sequences, according to the general principles of PRIMs theory, but that also allows efficient transfer from practice without combinatoric explosions in memory use or processing.

The solution I employed is as follows:

1. The agent proposes all known operators that can execute condition or action lines in the given instruction.

2. Preference rules prioritize operators that represent the largest known combination of condition or action lines, or choose randomly when there is a tie.

3. The agent iteratively retracts any proposals for operators that execute condition or action lines that have already been completed.

4. Once done executing all condition/action lines, increase the count of co-occurrence for all pairs of operator proposal rules and pairs of operator apply rules that were involved in the selected operators.

5. Once a co-occurrence count is past the chunking threshold, θ_p, chunk that pair of proposal or apply rules into a new rule that can be used in the future.

In other words, the PROP_1 agent does not chunk operator combinations until after they have co-occurred enough times to pass the θ_p threshold, and when the agent executes an instruction it uses only the largest possible operator combinations as soon as they are available. The agent then only combines together the fired rules that are used for the *selected operators*. It does not combine proposal rules that fired but which did not propose the selected operators.

By contrast, Actransfer/ACT-R combines any pair of co-occurring rules and adds them to procedural memory right away, but the agent is not likely to use its learned rules until after it gradually learns to increase their utility values, and even then it will select them with only gradually increasing probability. This means that Actransfer will have many more different mixtures of *redundant operation combinations*. For example, assume an agent is given an instruction for A, B, C, D, and has already practiced enough so that it must choose between executing either (AB) then (CD), or A then (BC) then D. An Actransfer agent might choose (AB) then (CD) one time, but because it selects probabilistically, the next time it might instead choose to perform A then (BC) then D. When it practices (AB) then (CD), it then learns the new rule ((AB)(CD)). When it later practices A then (BC) then D, it then learns the new rules (A(BC)) and ((BC)D). If it later practices these again, it will learn the rule (A(BC)D). The Actransfer agent will then have learned redundant rules for both ((AB)(CD)) and (A(BC)D) along with the full set of combinations that led to both paths: (AB), (BC), (CD), (A(BC)), and ((BC)D). This allows more breadth of rules that can be used or potentially transferred in the future, but many of them are unnecessary, and this approach does not scale well computationally if the number of valid operation combinations is large for a task or set of tasks.

The PROP_1 agent could practice either (AB) then (CD) or A then (BC) then D many times, and then eventually chunk either ((AB)(CD)) or (A(BC) or ((BC)D), but only one of these three. Whichever chunk it makes will depend on which combination is used most in the other instructions it practices over time. Assume it chunks (A(BC)). Then after more practice it will in total have learned only the combinations (AB), (BC), (CD), (A(BC)), and ((A(BC))D), and no more. This approach still allows transfer for the instructions actually used in practice without the large scaling footprint of Actransfer.

7.7 Evaluation

While the effort to define the computational details of PRIMs theory for Soar led to the theoretical contributions and distinctions listed above, there are two particular computational changes in the model that I need to evaluate.

- The effects of introducing PRIM resolution
- The effects of varying the chunking threshold, θ_p.

I can evaluate these two factors by toggling them on and off when running the model. As mentioned earlier, I can effectively turn off PRIM resolution and the corresponding layer of learning by providing the agent with the set of chunks that it would learn through PRIM resolution for its

specific task. If I disable PRIM resolution in this manner, I will describe this agent configuration with the term **`No PR`**.

I can also test the effects of varying θ_p, the number of times two rules must be seen together to be chunked into a new rule, by simply changing that parameter value.

These two factors are features I introduced to PRIMs theory with PROP_1, but there is another learning behavior that also exists in Actransfer that I can toggle in PROP_1, which is the last stage of PRIMs learning. This is the stage in which the agent compiles a single rule that carries out the practiced instruction automatically, that is, without retrieving the instruction. (This is the top layer in the hierarchy shown in Figure 4.4.) Actransfer always performs this final layer of learning, but subsequent experimental modifications to that architecture disabled it for computational reasons.[1] In later experimentation with PROP_2, which I discuss in chapter 8, I also found that disabling this final layer of learning improved the model's ability to match human behavior. For reasons that I also discuss in that chapter, I describe this setting using the term **`Auto`** for when this last learning level is enabled, and **`Deliberate`** for when it is disabled. Because this last learning step lets the agent perform task operations automatically without retrieving an instruction, it provides significant gains in performance speed relative to other steps of learning. Thus, I can predict that toggling this last layer of learning will significantly change the scale of agent performance.

I can predict that an `Auto` PROP_1 agent should be able to achieve more extreme gains in performance speed after practice than an Actransfer agent, because Soar will adopt learned rules instantly and consistently once they are learned, whereas Actransfer will adopt them gradually and probabilistically. It could be the case that PROP_1 performance would be more similar to Actransfer in `Deliberate` mode than `Auto` mode.

Beyond these factors that I can toggle in PROP_1, I can show the theoretical implications of the other differences with Actransfer without experimentation. My approach for retrieving instructions through WM knowledge rather than automatic architecture bias should have practically undetectable differences if implemented correctly, since both approaches have the same result that the agent retrieves the same sequence of instructions. The only difference might be a slight decrease in latency for the PROP_1 agent, since the Actransfer agent can technically have some wasted cycles from retrieving the wrong instructions from time to time due to random noise.

I was also interested in the consequences of how I made PROP_1 use deliberate co-occurrence counting to enact gradual learning, in contrast to ACT-R's mechanism for gradually adopting new rules. However, given that I had to use a different mechanism in order to support PRIMs learning in Soar at all, there is no way to simply toggle this kind of learning on and off in PROP_1 for the sake of comparison. Instead, I look to whether PROP_1 is capable of producing the same learning trends as Actransfer during experimentation. Still, one might predict that the PROP_1 agent will

[1] (N. Taatgen, personal communication, July 11, 2017.)

display more pronounced performance boosts from chunking, even if using this gradual approach, since it always uses newly composed rules in favor of more primitive components when possible, where Actransfer adopts new rules more probabilistically.

I also cannot simply toggle the way PROP_1 chunks sets of operators used together in a substate versus the way Actransfer composes sequential rules. There is no easy way to compare how Soar would perform with one alongside the other, since Soar does not natively support chunking sequential operators. However, I predict that the PROP_1 agent should be capable of slightly more transfer than Actransfer, since it has more pairs of operations to choose from at each stage of PRIM composition. Greater flexibility in the choice of which operations to combine should allow greater possibilities for achieving transfer.

All together, I predict that the main learning profiles of the architectures will be similar, including the amounts of transfer they provide, since they share the same core hierarchical learning theory from PRIMs. But I also predict that the absolute manner with which PROP_1 adopts newly learned rules will give it more absolute performance improvements from learning `Auto` rules. I also predict that PROP_1's ability to combine pairs of primitives from across sets of condition or action lines rather than sequences will produce more transfer. And finally I predict that a PROP_1 agent that uses PRIM resolution will have a higher initial performance cost and then steeper learning curve as it learns to use the necessary memory elements.

Testing these predictions, I gave a PROP_1 agent declarative instructions to perform in a simulation of the Elio (1986) arithmetic task experiment described in section 5.1. I copied the agent instruction logic and memory organization from what was used in the Actransfer simulation by Taatgen (2013), so that both model implementations learned to compose equivalent sequences of primitive operations.

The arithmetic task is well-suited for testing PROP_1, because it is a simple monotonic learning task that does not require too many complex reasoning strategies. This made it easier to isolate different learning behaviors for comparison between PROP_1 and Actransfer for this more constrained proof-of-concept stage of development.

A brief sweep of θ_p showed that values in the range of 16 to 24 provide comparable behavior to human and Actransfer results with PROP_1. If not otherwise noted, I use $\theta_p = 16$ below.

I ran two main experiments, as also described by Stearns et al. (2017). First, I toggled whether the agent performed PRIM resolution or not. Second, I evaluated the detailed effects of varying the chunking threshold θ_p.

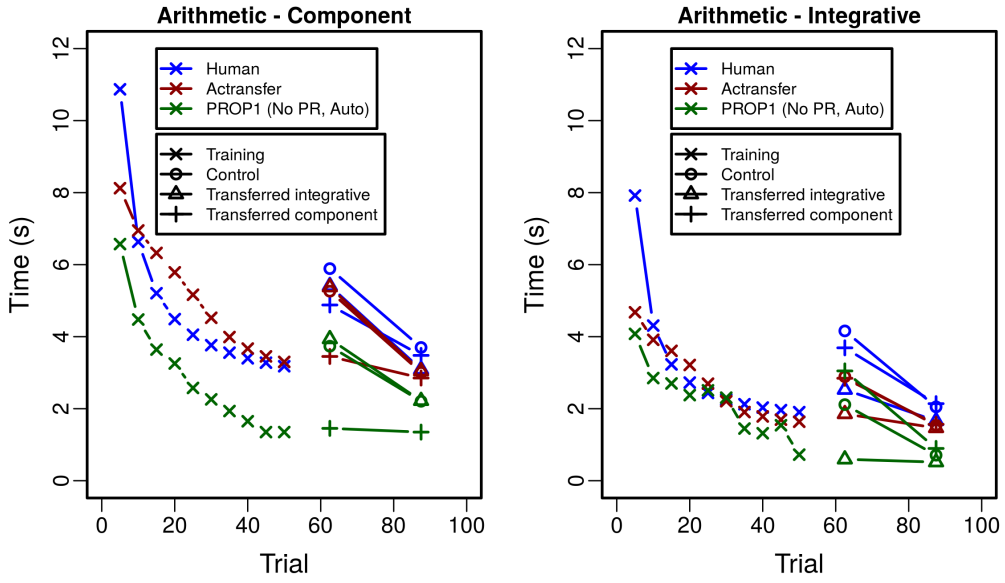
As described above, when plotting model behavior I omit latency that is due to agent decisions that support gradual learning, since these substitute for a necessary architectural mechanism that was not yet defined in Soar theory. I do, however, count latency for instruction retrievals. Because Soar does not include an approach for calculating the latency of declarative retrievals, I take these

numbers from Actransfer. Specifically, for each task step, I take the average retrieval time generated by the Actransfer agent and use that as the PROP$_1$ agent's retrieval time within that step. All data for the arithmetic task were averaged over 8 repetitions, as in Taatgen's originally reported results.

Actransfer follows ACT-R in assuming 50 msec per decision by default. I similarly use 50 msec per Soar decision. (This is true of all models discussed in this thesis.) Besides decision cycle time and declarative retrieval time, the Actransfer models also include time for visual/motor actions. In the arithmetic task, the Actransfer model used 0.3 sec to look up a single number from the prompt, 1 sec to find the max or min of a set of numbers, and 0.3 sec to enter a calculated number. I used the same times for PROPs.

7.7.1 Initial Replication Results

I show the initial results of my attempt to replicate Actransfer with PROP$_1$ before I show the results of my two experiments.



(a) Component steps

(b) Integrative steps

Figure 7.6: PROP$_1$ agent performance for the arithmetic task.

Figure 7.6 shows human, Actransfer, and PROP$_1$ performance for the arithmetic task. Again, on the left is the plot of time required to do component steps in the arithmetic algorithms, and on the right is the plot for integrative steps. Human performance is in blue, Actransfer in red, and PROP$_1$ in green. Here I use `No PR` and `Auto` for the PROP$_1$ agent, which is to say I disable PRIM

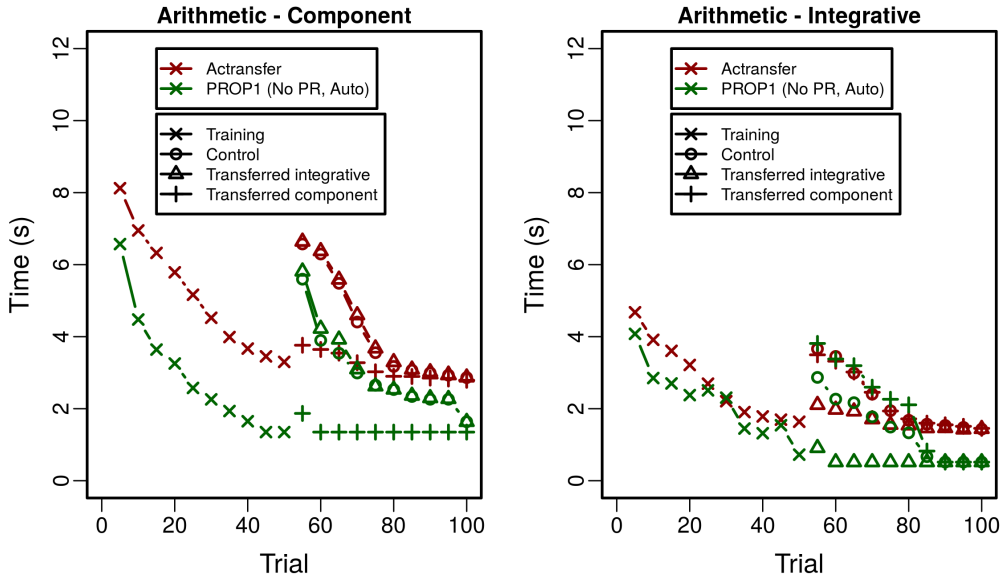
resolution and keep the final layer of learning. This is the configuration most similar to Actransfer in terms of learning theory.

As predicted, both models show similarly-shaped learning curves and transfer profiles, though the $PROP_1$ agent does perform faster than the Actransfer agent, particularly for component steps in the task. Figure 7.6a shows that the $PROP_1$ agent reaches optimal performance by the end of the 50 training trials. Optimal performance in this case occurs when all instructions are fully learned with final `Auto` rules. That the $PROP_1$ agent is able to level off at optimal performance where the Actransfer agent does not is also as predicted. As described earlier, this is due to the way that Actransfer is less able to take advantage of `Auto` rules after they are learned.

The relative speed increase is evident in Figure 7.6b for the integrative steps as well, though the gains in speed are less monotonic. With further inspection, the reason the agent performance jumps up and down is due to the affect of learned `Auto` rules on sequential instruction retrieval. As I described in section 7.3, $PROP_1$ keeps WM knowledge of where it is in its instruction sequence. Because `Auto` rules fire automatically without deliberate instruction retrieval, the agent does not know to update its WM knowledge of where it has progressed within the instruction sequence. Once the agent needs to rely on instructions again, it has to spend time searching through the sequence of its instructions to catch up to where its `Auto` rules had progressed in the task. The reason this results in the non-monotonic learning behavior for the integrative steps but not the component steps is due to the complexity of the instructions for those steps. Some of instructions for the integrative steps that are used in sequence have fewer condition and action lines than other steps. This means that the agent can compile `Auto` rules for each of these with fewer iterations of practice than it requires for other steps. When the agent learns `Auto` rules for one block of instructions faster than the rest, the agent is able to skip through a large chunk of the task at once when at the learned block, but it then has to slow down when it reaches a block of instructions it has not yet learned in order for its WM representation to catch up. The component steps, by contrast, are more uniform in complexity, such that the agent can compile those instructions at about the same rate.

In Elio (1986)'s original publication, the human transfer data over trials 51-100 were averaged into two points, as I also show above for Actransfer and $PROP_1$. But I am able to examine the Actransfer and $PROP_1$ data for trials 51-100 in greater detail. This is shown in Figure 7.7.

This figure better shows the relative transfer effects for the different transfer conditions. Each of the first data points in the transfer conditions is lower than the first data point in the training period, though not by much for two of the three transfer conditions. In the rest of this thesis, I show the averaged transfer data as before, to better compare with human data as well as to improve readability in the figures.



(a) Component steps (b) Integrative steps

Figure 7.7: PROP$_1$ agent performance for the arithmetic task with higher resolution for transfer data.

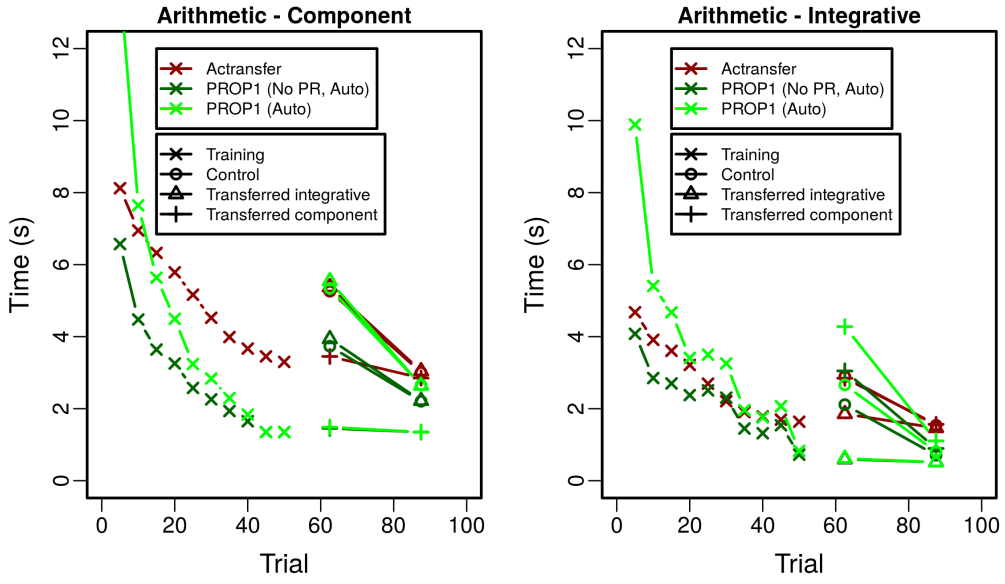
7.7.2 Experiment 1: Effects of PRIM Resolution

I now describe my first experiment, in which I test the effects of PRIM resolution.

Recall that PRIM resolution at first requires subgoal processing to find the WM references needed for each condition and action line. The prediction is that, this extra processing should slow down agent initially, but that it should quickly catch up as the agent learns chunks for primitive memory operations on specific WM locations.

Figure 7.8 shows this to be the case. This figure shows the same Actransfer and PROP$_1$ agents as above in Figure 7.6, with the addition of the plots for the agent that has PRIM resolution enabled, shown in light green. I observe two main effects of adding PRIM resolution. First is that the overall scale of task latency is stretched vertically to be longer. Second is that, as predicted, the amount of latency reduction in this stretching is disproportionately higher for the early part of the learning curve. Note that the first three data points on the light green plot are much further from those of the dark green plot in the training case than are the following points. Both PROP$_1$ plots converge to the same optimal performance as they learn `Auto` rules for the task. To better see the effect of PRIM resolution, I disable `Auto` rules, as shown in Figure 7.9.

Figure 7.9 shows the PROP$_1$ agents with and without PRIM resolution, both set to `Deliberate` rather than `Auto`. This more clearly shows that the learning curves with PRIM resolution are practically parallel to those without it except for the first few data points. Intuitively, this makes sense.



(a) Component steps

(b) Integrative steps

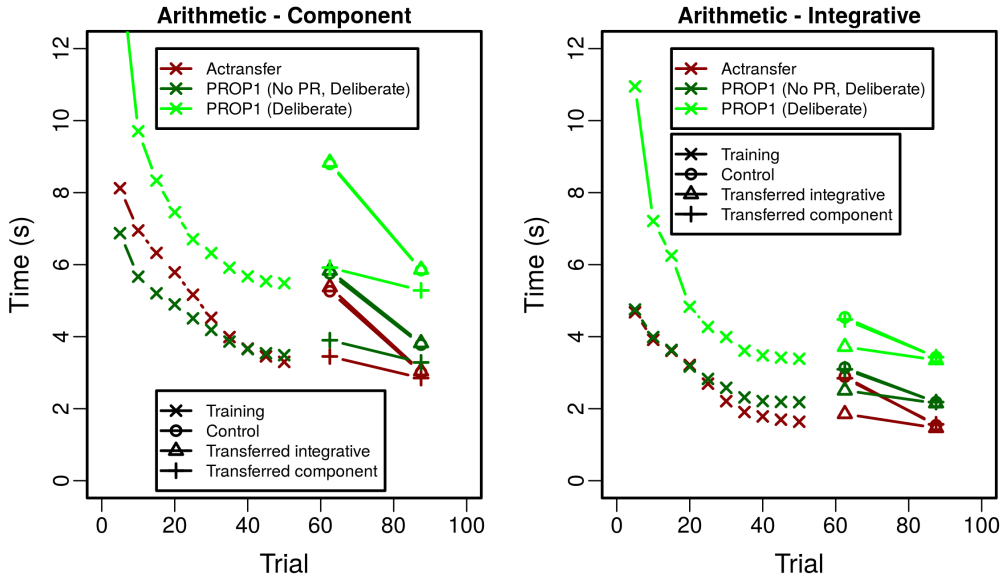
Figure 7.8: PROP$_1$ models with and without PRIM Resolution (PR).

PRIM resolution provides an extra layer of processing and learning that comes at the start of task practice only.

There are a few notable takeaways from these results. Firstly, PROP$_1$ with PRIM resolution results in a much more human-like power-law learning curve for training trials 1-50. This was not my aim when I introduced PRIM resolution, but it supports the addition of this deeper primitive layer of processing in the model. Secondly, PROP$_1$ set to `Deliberate` is indeed more aligned with the original Actransfer model. However, Actransfer achieved a slightly lower latency than PROP$_1$ `Deliberate` can by the end of the trials, since it can learn and eventually use `Auto` rules. Thirdly, the way I modified P2 in PROP$_1$ so that the agent relies on WM knowledge to select instructions from SMEM introduces the possibility of non-monotonic behavior in the learning curve. This was unexpected and produces different behavior from Actransfer. A non-smooth learning curve is within the realm of realistic human behavior for an individual. The smooth power-law curve of the human data here is taken from averaging the results of all human participants, and the data for individuals is not available. Though model data is averaged over the course of 8 repetitions, there is little room for variation or randomness in the PROP$_1$ model. Effectively, the model data shown is that of a single individual.

There also appears to be a constant vertical shift in performance when PRIM resolution is enabled. With further inspection of the agent behavior, this is due to an increase in processing overheads that comes with supporting PRIM resolution in the PROP$_1$ implementation.

Figure 7.10 shows the PROP$_1$ (`Deliberate`) model alongside Actransfer and human perfor-



(a) Component steps

(b) Integrative steps

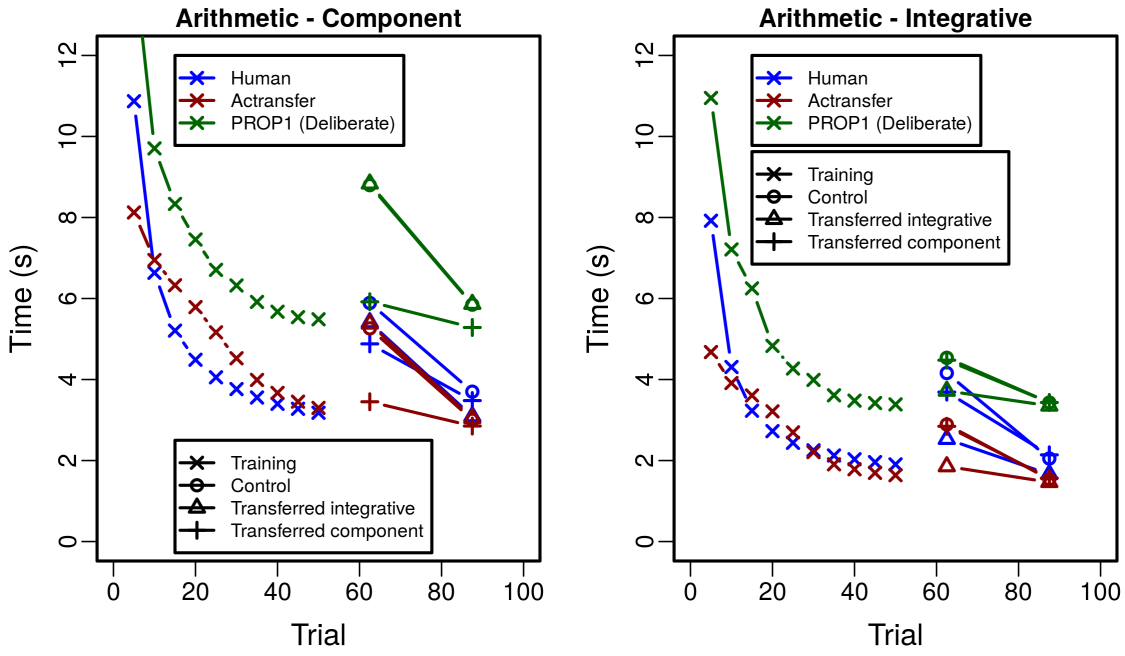
Figure 7.9: The PROP$_1$ models without the final `Auto` rule learning stage, with and without PRIM Resolution (PR).

mance for comparison. The similarity in training trial learning curves between human data and the PROP$_1$ model is striking, though reaction time is shifted up in the PROP$_1$ model.

	Component Steps			**Integrative Steps**		
Model	r^2	*MAE*	*MAPE*	r^2	*MAE*	*MAPE*
Actransfer	0.808	0.84	15.7	0.758	**0.58**	**14.2**
PROP$_1$ (No PR, Auto)	0.941	1.94	41.9	0.737	0.93	26.7
PROP$_1$ (Auto)	0.988	1.28	29.4	0.937	0.89	29.2
PROP$_1$ (No PR, Deliberate)	0.898	**0.74**	**11.7**	0.838	0.59	15.1
PROP$_1$ (Deliberate)	**0.994**	2.71	61.3	**0.969**	2.05	72.2

Table 7.2: PROP$_1$ goodness-of-fit measures for arithmetic data. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

I use two kinds of quantitative goodness-of-fit measures to compare model results with human performance. I first use r^2 to measure how well each model produces the same shape of learning curve as humans. This is particularly lets me measure how well the model's overall learning and transfer mirrors human processing. It does not measure whether the data points are close to human data, however, only whether the shape of learning and transfer is the same. Therefore, I also use both Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) as goodness-of-fit measures. MAE is the average of the differences between the model and human data, $\frac{\sum_{i=1}^{n} |h_i - m_i|}{n}$, where h_i and m_i are human and model data points, respectively. MAE measures



(a) Component steps

(b) Integrative steps

Figure 7.10: The best r^2 fit PROP$_1$ model compared with both human and Actransfer performance.

error in the same scale as the data being measured. For a more normalized measure, MAPE instead shows the error between model and human performance as a percentage of the human performance, $\frac{100}{n}\sum_{i=1}^{n}\frac{|h_i-m_i|}{h_i}$. This second kind of measure does not directly show how well the shape of learning curve mirrors humans, but it does measure the closeness of the data points to human performance on average.

Table 7.2 shows these goodness-of-fit measures for the arithmetic task training trials. The model with the closest fit to humans is shown in bold in each column. PROP$_1$ (`Deliberate`) has the best r^2 measure for both the component and integrative steps. This shows that this model does indeed mirror the shape of the human learning curve better than Actransfer or alternate PROP$_1$ models. However, PROP$_1$ (`Deliberate`) also has the worst MAE and MAPE results, due to the vertical shift in the baseline of performance.

The similar shape but upward shift in PROP$_1$ (`Deliberate`) performance compared to humans implies the learning approach of PROP$_1$ is good, but that there is a processing overhead in PROP$_1$ that a better model would not include. The implication of unnecessary processing is not surprising, given how much of PROP$_1$'s processing uses agent decision cycles to simulate Actransfer's architectural processing. I show in sections 8.4 that when I develop the PROPs system further to be more consistent with the surrounding Soar architecture, the upward shift is reduced while the shape is maintained. In section 9.4 I show that incorporating Soar's theory for parallel proposal

rules into PRIMs processing makes the shift go away entirely.

7.7.3 Transfer

Elio (1986) originally measured transfer as the difference between the average performance in the control condition and the transferred integrative or transferred component conditions. I report this same metric for both Actransfer and the PROP$_1$ models. However, because each model shows a different scale in its learning curve, it is difficult to compare these values meaningfully. I therefore add an additional metric for transfer, which is the percent of transfer in each transfer condition relative to its preceding training condition. The specific formula I use is $(train_{start} - transfer_{start})/(train_{start} - train_{end}) \times 100$. I use the average of the first 25 trials in the training data for $train_{start}$ to match the transfer condition data. For $train_{end}$, I use the value at trial 50 in the training data. A score of 100% means that the agent started the transfer task with the same average performance it had achieved by the end of the training task. A score of 0% means the agent starts the transfer task with the same average performance it had at the start of the training task. Because the starting inputs are averaged, an agent could get a score greater than 100% if it continues learning during the first 25 trials of a transfer condition beyond the level of skill it had achieved by the end of training. Negative transfer is also possible if the agent begins a transfer condition with worse average performance than it had in training.

The benefit of my percentage metric is that it measures transfer relative to each model's learning, and is invariant to differences in scale across different models. However, meaningful interpretation of this metric assumes that a model's learning curve for a transfer condition looks identical to that of the training condition if it had not been preceded by training. In Elio's experiment, the calculations for the training and transfer conditions were drawn from the same pool of mathematical operations and allocated to training or transfer conditions randomly per subject, and therefore this assumption should hold as a close approximation. True transfer percentages require measuring the transfer conditions without preceding them with training. This was missing from the human data, however.

All transfer scores are shown in tables 7.3 and 7.4. The table columns separate transfer scores for control, integrative, and component transfer conditions. The top half of each table shows the "Difference from Control" score, as used by Elio (1986). The lower half shows my alternate "Percent Transfer" measure. The transfer percents for each model are followed in parentheses by the difference from the human percent transfer. The original human transfer is shown at the top of each section for each measure. For instance, in Table 7.3, the percent transfer for humans in the control condition is 11.75%. The percent transfer for PROP$_1$ (`Deliberate`) in that condition is 13.50%, which is a difference of +1.75%. Models with percent transfer closest to humans are

marked in bold. The PROP$_1$ (`No PR, Auto`) model has the smallest difference in the control condition from human transfer at +1.64%, and so it is marked in bold in that column.

Component Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		0.48	1.01
Actransfer		-0.11	1.81
PROP$_1$ (No PR, Auto)		-0.20	2.28
PROP$_1$ (Auto)		-0.20	3.86
PROP$_1$ (No PR, Deliberate)		-0.08	1.86
PROP$_1$ (Deliberate)		-0.04	2.88
	Percent Transfer		
Human	11.75%	27.38%	44.63%
Actransfer	38.09% (+26.34)	34.51% (+7.13)	95.23% (+50.60)
PROP$_1$ (No PR, Auto)	**13.39% (+1.64)**	6.13% (-21.25)	96.23% (+51.60)
PROP$_1$ (Auto)	29.02% (+17.27)	**25.46% (-1.92)**	97.58% (+52.95)
PROP$_1$ (No PR, Deliberate)	-16.89% (-28.63)	-20.98% (-48.36)	**78.66% (+34.02)**
PROP$_1$ (Deliberate)	13.50% (+1.75)	12.55% (-14.82)	88.72% (+44.08)

Table 7.3: PROP$_1$ transfer for arithmetic component steps.

Overall, transfer trends are similar across models. Both Actransfer and PROP$_1$ models achieve roughly double transfer than humans in the transferred component condition for the component steps and in the transferred integrative condition for the integrative steps. This indicates that there are more nuances in the transferred knowledge for humans than the models capture. This is likely a property of the written task instructions, though it is difficult to draw conclusions.

Notably, the PROP$_1$ (`No PR, Deliberate`) agent showed *negative* transfer in the control and integrative transfer conditions. That is, its initial performance in these was poorer than in the initial training trials. With inspection, this is due to the LTDM retrieval time taken from Actransfer combined with a relatively greater number of retrievals done by the PROP$_1$ agent compared to the Actransfer agent. In Actransfer, the average initial time per retrieval for these transfer conditions is about 0.10-0.14 sec, while the average initial retrieval time for the training is about 0.06-0.07 sec. In PROP$_1$, the decrease in the number of decision cycles that comes with procedural learning for these cases is not enough to compensate for this doubling in retrieval time, and thus the overall effect is worse performance.

Relative to humans, there is no clear winner for which model has the most similar percentage transfer. For component steps (Table 7.3), different PROP$_1$ models are closest for each transfer condition, though among the PROP$_1$ model variants, PROP$_1$ (`Deliberate`) comes in second for all transfer conditions. For integrative steps (Table 7.4), the PROP$_1$ (`No PR, Deliberate`)

Integrative Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		1.63	0.47
Actransfer		1.03	0.04
$PROP_1$ (No PR, Auto)		1.52	-0.94
$PROP_1$ (Auto)		2.05	-1.62
$PROP_1$ (No PR, Deliberate)		0.63	0.04
$PROP_1$ (Deliberate)		0.82	0.06
	Percent Transfer		
Human	-1.62%	71.84%	19.56%
Actransfer	36.89% (+38.51)	89.07% (+17.23)	39.14% (+19.57)
$PROP_1$ (No PR, Auto)	36.26% (+37.88)	105.88% (+34.04)	-6.69% (-26.25)
$PROP_1$ (Auto)	59.70% (+61.32)	104.70% (+32.86)	**24.15% (+4.58)**
$PROP_1$ (No PR, Deliberate)	**35.88% (+37.50)**	**78.12% (+6.28)**	38.80% (+19.24)
$PROP_1$ (Deliberate)	65.34% (+66.96)	89.95% (+18.11)	67.10% (+47.54)

Table 7.4: $PROP_1$ transfer for arithmetic integrative steps

model is closest for two of the three transfer conditions.

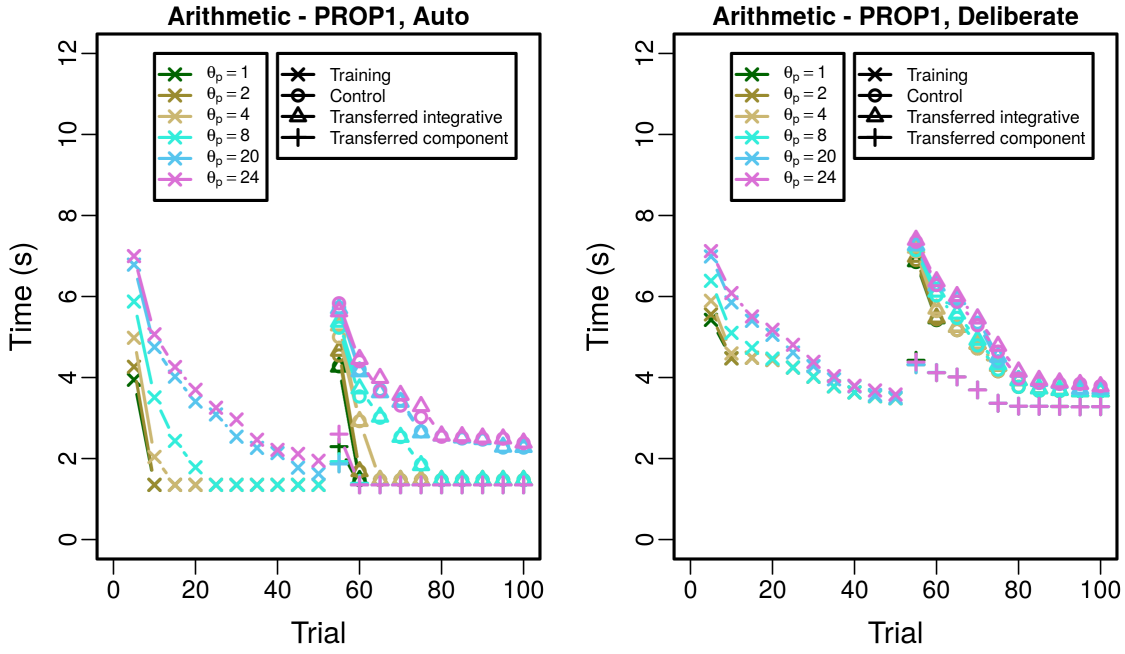
7.7.4 Experiment 2: Effects of θ_p

To test the effects of the θ_p chunking threshold, I performed a sweep across different values from $\theta_p = 1$ to $\theta_p = 24$. Figure 7.11 shows this sweep for component steps for both `No PR, Auto` and `No PR, Deliberate` models.

Note how in Figure 7.11a the first data point for the transferred component condition (the first plus sign) is the same for all values of θ_p except for $\theta_p = 1$ (dark green) and $\theta_p = 24$ (pink). The first data points for the other transfer conditions by contrast gradually increase in value with higher θ_p. Similarly, in Figure 7.11b, all first data points for the transferred component condition remain the same despite the increase in θ_p.

The fact that going from $\theta_p = 1$ to $\theta_p = 2$ *decreases* latency for the transferred component condition in Figure 7.11a shows that higher values of θ_p can allow greater transfer, as hypothesized in section 7.5. The higher threshold allows the agent to practice more varieties of PRIM combinations, and thus to find which combinations will be most transferable. However, there are no further gains in transfer in this task from increasing θ_p higher than 2. Beyond that point, increasing θ_p only increases the learning time.

The fact that even an order of magnitude increase in θ_p from 2 to 20 is not enough to increase the learning time for the transferred component condition indicates that the gains from transfer are great enough to render the higher threshold irrelevant, at least until $\theta_p = 24$ in the `Auto` model.



(a) Component, Auto, No PR

(b) Component, Deliberate, No PR

Figure 7.11: The change in transfer in the arithmetic task with increase in θ_p.

Though $\theta_p = 2$ was sufficient for maximal transfer in this task, the proper value for maximal transfer in general is going to be dependent on both the variety of PRIMs within practiced rules and on the variety of rules practiced across tasks. The greater the variety of PRIMs within practiced rules, the more options there are for combining them, the more experience the agent needs to determine which combinations are optimal. The order of practiced task instructions will also affect the optimal θ_p. For example, if an agent repeatedly practices one instruction twenty times in a row before then practicing second twenty times, the agent cannot incorporate any experience from the second within its procedural learning unless its θ_p is greater than twenty. That is, it must wait to compile procedures until after it has started practicing the second. However, if the agent alternates back and forth between both instructions every other attempt rather than twenty attempts at a time, θ_p could be as low as two. Since the optimal θ_p in this task is two, this implies that, in the time it takes for the agent to experience a combination of PRIMs two times, it will have experienced all the other relevant combinations of PRIMs that it needs to consider.

An investigation of the processing that led to these results showed that much of the transfer benefits of increasing $\theta_p = 1$ to $\theta_p = 2$ came because of PROP$_1$'s use of *sets* rather than *sequences* when representing instruction lines. The set representation allows the architecture to potentially combine any pair PRIMs used for an instruction group of conditions or actions, while the sequence representation only allows pairs of adjacent conditions and adjacent actions. The more possible

pairings of the set representation thus increases the benefit of gradual learning so that the architecture can gather more experience for which pairs might transfer most.

7.8 Discussion

With $PROP_1$ I replicated the processes of Actransfer in Soar, and made a few modifications to the computational theory to be compatible with the Soar architecture. A significant aspect of this work was not only to make the computation compatible with Soar, however, but also to wrestle with the theory of what exactly is a PRIM? Though it has a clear computational meaning in the context of ACT-R and Actransfer, as a primitive rule, this meaning does not transfer cleanly to Soar, because rules in Soar represent different processing.

I introduced PRIM resolution to support P1 in Soar's unbounded graph memory space, and I showed that this leads to a more pronounced power-law learning curve. The $PROP_1$ models that include PRIM resolution provide the best fits the human data in terms of shape. Omitting PRIM resolution, however, roughly emulates Actransfer's learning profile. At the very least, this implies that an additional type of primitive processing and an extra layer of learning is desirable for modeling humans.

$PROP_1$ extends PRIMs theory by providing an explanation for where the PRIM rules of Actransfer come from. It posits that innate, general processing can apply PRIM operations to any WM element address, and that this application is *learned* to form the specific PRIM rules used in Actransfer. The shape of the model's performance matches human subjects when it uses this processing in this learning at the start of this task.

Independent of Soar's unbounded memory, PRIM resolution allows a level of rules that are even simpler than Actransfer PRIMs that leads to better learning performance than the PRIMS of Actransfer. Thus, applying PRIM resolution to Actransfer with its fixed slots ought to improve its ability to match human performance as well. I know of no reason that PRIM resolution could not be implemented in Actransfer. It is necessary to support PRIMs theory with Soar's design of WM, but ACT-R's rules can also use variables like Soar's, and its WM space is no more complex than Soar's. Actransfer's WM could even be entirely implemented within Soar's as a constrained type of WM graph, where each buffer is its own node in the graph with a child WME for each slot. (This is similar to the example WM graph shown in Figure 6.2.)

Adding PRIM resolution to Actransfer would also allow it to be consistent with ACT-R's standing design for WM while also greatly simplifying the initialization of procedural memory. Actransfer would require additional reserved WM locations to cache the memory references it would find with PRIM resolution. It could then have generic PRIM rules per operation type that bind variable values to the references held in this cache.

PRIM resolution was necessary in Soar not just because Soar's WM is unbounded, but also because it is *hierarchical*. If Soar's WM was unbounded but flat, where all WMEs were held under the root state node, then no search would be required to access them. But no innate rule can directly access the items kept at an unbounded depth (with variable structure) without first traversing the graph to find them. PRIM resolution is a general way of learning more direct ways of accessing memory that suit whichever hierarchical WM structure is used by a subject.

An open question for PRIMs theory is what the most primitive layer of knowledge ought to correspond with in humans. Why, for instance, would adult subjects in these tasks still need to learn to compile skills for individual PRIMs if they were, indeed, primitive? Shouldn't adults have had enough experience before beginning an experimental task to have already compiled a great deal of procedural knowledge beyond the primitive level, which could be transferred to each task from the start? It could be that in humans the relevant set of WM "addresses" is unique per task. This would support the need for PRIM resolution when learning a new task. It could also be that the kind of knowledge that is called "primitive" in PRIMs theory is at a level of abstraction that obscures an even more primitive layer that humans would build up during their lifetime, and that the kind of learning demonstrated with PRIMs should be considered a more short-term or context-specific sort of learning. This would go against the classic ACT-R theory notion that production rules correspond with permanent basal ganglia procedures. However, there is some recent research suggests the theory that production rules in cognitive architectures ought to be considered to correspond with different brain regions depending on their function (Rice & Stocco, 2018). I discuss the question of developmental and adult learning for PRIMs further in chapter 10.

Future work might explore how PRIM resolution affects computational load of using procedural memory. In theory, it should improve it considerably compared to the Actransfer approach, since it reduces the number of potentially matching rules in procedural memory.

I also added the distinction between primitive rules (PRIMs) and primitive operators (PROPs). This incorporates part of Soar theory, but does not noticeably affect the agent's task performance.

The PROP_1 model with the most similar learning curve to humans in terms of overall shape was the `Deliberate` model. As I will discuss in greater detail in the next chapter, this supports the view that subjects in this task do not ever proceduralize it so far that they can carry it out completely autonomously without loading declarative task knowledge into WM. This might not be the case for all tasks. More physical skills such as running or martial arts are known to be cases where highly-trained humans can act without much declarative awareness of effort. Any arithmetic algorithm with multiple steps, however, will almost certainly always require some level of cognitive awareness and control.

I designed the PROP_1 agent to control P2 retrieval selection using a rote retrieval sequence kept in WM and not through automatic architectural condition checking. Like Actransfer's solution for

P2, this allows the model to function (and replicate Actransfer), but it is not meant to provide a detailed cognitive model of P2 processing.

I built the PROP_1 agent to chunk pairs of instruction lines together gradually by having the agent deliberately count up each occurrence of a potential chunk until it had seen it θ_p times. Only after the agent noticed through its reasoning that the count had passed the θ_p threshold would it let the architecture chunk the pair together. This substituted for a lack of Soar support for gradual chunking. An actual model of subject reasoning in Soar should never include such decision making as a means of controlling an architectural process. I therefore omitted the decision cycle time for this process from the final model results. Even so, removing the need for this processing was a high priority for me in future iterations of PROPs.

It was difficult for me to define gradual learning in PROP_1 for Soar, because the original publication of PRIMs theory assumes the ACT-R mechanism for gradual rule learning. I therefore emulated ACT-R's mechanism. However, is this assumption necessary for implementing the theory in other architectures? Ultimately, Soar's procedural learning is simply very different from that of ACT-R. That difference required the agent to control its architectural processing. The contrast between Soar and ACT-R learning computation theory becomes more apparent as I discuss subsequent iterations of the PROPs system, since my work in those iterations was to progressively adapt the system to reflect Soar theory and incorporate the features it can contribute.

I designed the PROP_1 agent to compile instruction lines by compiling pairs from the *set* of lines in the instruction rather than from only the sequentially adjacent lines. I was not able to easily toggle this feature on an off for experimentation due to the fact that Soar naturally favors a set representation, and a sequential representation would require a fundamentally different and custom-tailored agent constructed differently from the ground up. However, as stated, this difference contributed to the pronounced transfer benefit of increasing $\theta_p = 1$ to $\theta_p = 2$. Otherwise, I do not observe a significant net effect in agent performance, though the number of variables at play, such as θ_p itself, make such a determination difficult.

A θ_p as low as 2 was sufficient to achieve optimal within-task transfer for the simple arithmetic task. But it would be expected that the more complex the tasks and the longer the span of learning for an agent, the higher this threshold ought to be. Future work might investigate what kind of formal relation might be found between the optimal chunking threshold and the ordering and complexity of different practiced procedures. This could then be compared with what is known about how practice structure affects transfer in humans (see (National Research Council, 1994) for discussion).

My work with PROP_1 showed that despite differences between ACT-R and Soar models of WM and learning, PRIMs theory can be implemented in both to achieve similar results. In so doing, I introduced the PRIM resolution approach for information processing in unbounded memory spaces

like Soar's. Because this model defines primitive rules that support task-independent processing in a way that is consistent with Soar theory, $PROP_1$ addresses the P1 implementation gap.

For additional discussion on how the definition of a PRIM relates to how an architecture defines "working memory," see appendix B.

CHAPTER 8

PROPs Iteration 2: Defining Declarative Retrievals

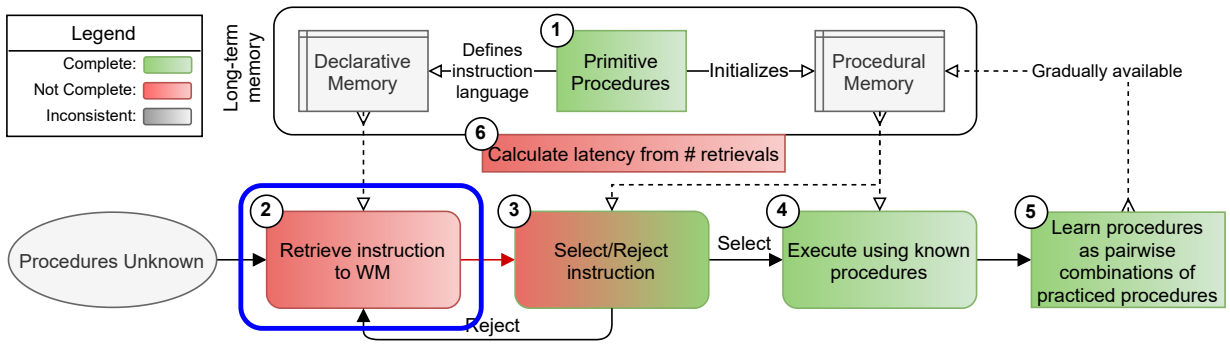


Figure 8.1: The $PROP_1$ flow diagram. $PROP_2$ addresses incomplete computation circled in blue.

Figure 8.1 shows the PRIMs flow diagram colored according to the completion level of $PROP_1$. Once I had established $PROP_1$ as a working replication of Actransfer's main functions using Soar, I began building $PROP_2$ to address computational gaps in the procedural learning pipeline, specifically the P2 *retrieval selection* phase, as circled in blue in the figure. In addition to implementing P2 for PRIMs theory, I also set out to define gradual procedural learning for Soar, a capability that was missing from that architecture, as I described in the previous section.

Thus, $PROP_2$ specifically targeted the following two problems:

1. The Actransfer model does not define a computational approach for selecting an instruction during retrievals (P2).

2. Soar does not define gradual chunking.

As I described in chapter 4, PRIMs theory does not define why the agent might retrieve an instruction that is already likely to have satisfied conditions. Actransfer used an architectural preprocessing step that would analyze all instructions and their conditions prior to every retrieval to ensure that usable instructions were likely to be selected. This allowed enough P2 functionality

so that the transfer mechanics of Actransfer could be evaluated, but otherwise represents a significant gap in the overall computational pipeline and theory. The $PROP_1$ approach for guiding retrievals was similarly only meant to be a simple functional substitute that allowed the agent to run. A properly-defined computational approach that attempted to address cognitive theory was still missing.

I saw two possible solutions for P2 that might work in Soar. First was Taatgen's hypothesis that a spreading activation mechanism might define the retrieval process. At the time of finishing $PROP_1$, a spreading activation mechanism had been recently added to Soar (S. J. M. Jones et al., 2016), though its capabilities were largely untested. Second was a more classic Soar approach in which agent decision making could directly influence retrievals. As I described under step 3.b.2 of my methodology, since both options were plausible, I set out to implement and evaluate both and compare their merits. $PROP_2$ attempted the first possible solution: using spreading activation.

In chapter 7 I introduced the second problem that I address with $PROP_2$, how to define gradual procedural learning. As I described in chapter 6, Soar rules are always used whenever their conditions are satisfied. This makes Actransfer's ACT-R-based approach for gradually adopting the use of a learned rule impossible for Soar without a significant change in Soar theory.

For this problem, I took the experimental approach of $PROP_1$ that used deliberate agent decision making to simulate gradual chunking and implemented it as an architectural mechanism in Soar. I then restructured the agent design accordingly to leverage this new gradual chunking mechanism.

8.1 Problem 1: Retrieval Selection

My work with $PROP_2$ to define retrieval selection using spreading activation had three parts, to identify answers to the following three questions:

1. What knowledge structures and/or rules does an agent need for spreading activation to provide desired PRIMs theory behavior?

2. What would be required for these knowledge structures and/or rules to be learned as part of general skill learning?

3. What do the answers to the above computational questions imply for cognitive theory?

I describe my explorations of each of these questions below.

8.1.1 Using Spreading Activation

What would it take to use spreading activation to guide the instruction retrieval process in Soar? I review the problem, the computational theory of Soar, and my solution.

The Problem For PRIMs theory to be practical for human modeling, any instruction that the agent retrieves into WM ought to already be very likely to have satisfied condition lines. Spreading activation would need to alter the relative activations of different instructions in LTDM so that instructions with satisfied conditions are the most likely to be retrieved.

How likely must it be that *all* conditions in the retrieved instruction are satisfied? If the agent retrieves an instruction and even one condition is not satisfied, then the agent must discard them and attempt another retrieval, and this takes time. For the tasks involved in Actransfer's experiment suite, a model of human performance cannot afford to reject instructions except on rare occasions. This is because these tasks, particularly those for the Stroop and task-switching experiments, require stimulus-response reaction times on the order of fractions of a second. If instruction retrievals can take anywhere from 0.3-0.9 sec, as is the model for some Actransfer agents, then multiple retrievals for instructions would quickly prevent human-like response times for these tasks.

Soar Computation In this thesis I aim to fill in the gaps in Actransfer's theory using Soar's cognitive architecture computation. How does spreading activation work in Soar?

First, the main function of activation in Soar is to break ties when there are multiple possible results for a query to SMEM. When this happens the agent retrieves the memory with the highest activation. Second, spreading in Soar works by increasing the activations of memories associated with the contents of WM. Any long-term memories that are active in WM boost the activations of other long-term memories to which they are associated. For example, consider Figure 8.2 below. On the left of the figure is SMEM. On the right is WM. There are three long-term memories currently active in WM, labeled N1, N2, and N6, shown as gray circles. Because these are in WM, they get an activation boost, shown via thicker black circles in SMEM. In SMEM, these memories are connected to other long-term memories, specifically N4, N5, and N8. Soar spreading activation boosts the activations of those connected memories. Soar normalizes the amount of spread given by a memory across its descendants (the fan effect). In this case, N2 gives less boost to each of N4 and N5 because it divides the total amount of spread between them, whereas N1 gives its full boost straight to N4. Any spread given to these immediate descendants continues to spread to their descendants. The amount of spread decays with distance. In this case, N4 passes some spread along to N7 and N8, and N5 also passes some along to N8. Because N8 receives spread from N6 as well as decayed spread from N1 and N2, it has the highest total activation in this example. If the agent did a retrieval for one of these memories, it would retrieve N8 into WM.

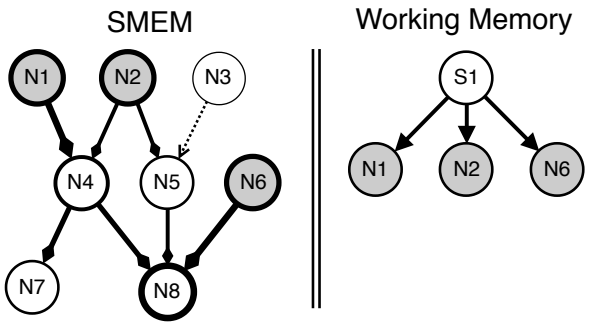


Figure 8.2: Spreading activation in Soar. Gray circles depict memories from semantic memory (SMEM) that are currently active in working memory. Line thickness on circles in SMEM depicts the activation of those memories. Arrow thickness in SMEM represents the relative strength of activation spreading from those memories.

To use such a mechanism for the goals of P2 in PRIMs theory is non-trivial. Preliminary experimentation showed that if the instruction activation is defined as a simple linear function of the number of satisfied conditions (add a constant amount of activation for each satisfied condition), then it is easy to reach scenarios where instructions with fully-satisfied conditions never have the highest activation. Imagine two instructions, one with conditions A,B,C,D and another with only conditions A,B. If only conditions A,B,C are actually satisfied in WM, then the first instruction would receive more activation boost than the second, even though the first still has an unsatisfied condition while the second does not.

Because instructions should ideally be retrieved only if all their conditions are satisfied, and because activation for biasing retrievals must spread from knowledge that is explicitly in WM, this implies that the agent must have *declarative knowledge* of satisfied conditions in WM in some form. No other approach would let satisfied conditions define the results of spreading activation. So an additional question is how can SMEM memory elements related to conditions be placed into WM in order to initiate spreading? In Soar, this can happen in only two ways: SMEM memories can be retrieved into WM through a deliberate retrieval or query action, or they can be created in WM through a chunk that recreates a past retrieval.[1] Somehow, the agent must obtain WM knowledge of satisfied conditions before it goes through the effort of retrieving an instruction that uses those conditions, and do this without compromising human-like speed in task performance.

8.1.2 Approach

My approach is depicted in Figure 8.3. The figure shows my representation in SMEM for two instructions and their associated spreading structures, labeled "Skill 1" and "Skill 2." The top row of circles, labeled "Knowledge of satisfied conditions" represents declarative knowledge of

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

conditions that can be retrieved into WM. (I will discuss how these can be created in WM shortly.) The next two rows labeled "Knowledge of rule condition structure" represent SMEM structures that normalize the spreading effect so that the agent is more likely to only retrieve instructions with all conditions satisfied. The lower two circles labeled "Instructions," I1 and I2, and the connected "PROP conditions and actions" are the actual instructions that the agent must retrieve to perform a task operation. Skill 1 has four conditions, labeled P1, P2, P3, and P4, which correspond with C1, C2, C3, and C4. Skill 2 has only two conditions, P5 and P6, corresponding also with C5 and C6. P7 at the bottom could represent an action primitive that is shared by both instructions, but is irrelevant for this example. Note the distinction between the PROP condition structures shown at the bottom, which describe the rule conditions, and the C1-C6 structures at the top, which describe whether particular conditions are *satisfied*. In the figure, three of four conditions are known to be satisfied for Skill 1, and two of two conditions are satisfied for Skill 2. The depicted structure takes advantage of the way Soar normalizes spread to ensure that I1 thus gets 3/4 of a full activation boost, while I2 gets a full 2/2 boost.

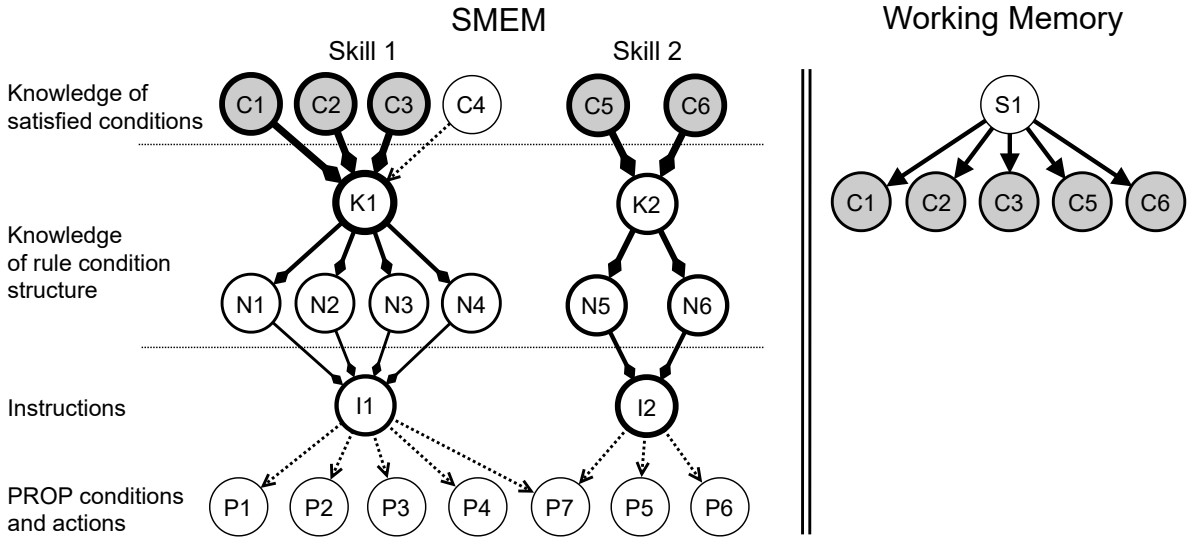


Figure 8.3: Example $PROP_2$ SMEM structures for spreading activation. Nodes in gray are present in working memory. Line thickness in SMEM indicates activation strength. Spread from these is normalized over the number of a rule's conditions.

Normalizing Spread The "Knowledge of skill condition structure" layer in Figure 8.3 uses the Soar approach for normalizing spread across a node's children in order to also normalize the amount of spread any instructions receive from satisfied conditions. In the figure, any spread that passes through node K1 will be divided by four, because it has four child nodes. Any spread that passes through K2 will be divided by two. Thus, with three of four conditions satisfied, the instruction for Skill 1 (I1) receives 3/4 normalized spread, and, with two of two conditions satisfied,

the instruction for Skill 2 (I2) receives 2/2 spread. I2 therefore has the greater activation, and will be preferred during instruction retrieval. The approach of the SMEM structure shown in the figure is engineered, but follows directly from the theory of Soar's spreading normalization.

Learning to Recognize Satisfied Conditions For spreading to activate instructions appropriately, the $PROP_2$ agent must create or remove long-term memory elements within WM, causing spread, whenever conditions for instructions become satisfied or stop being satisfied. How can the agent do this? Doing a deliberate SMEM retrieval to update which conditions were considered satisfied *every* time WM changed would take far too much modeling time to allow human-like performance. But if the agent creates SMEM memories in WM via chunks, then potentially no time cost is required.

I therefore design the $PROP_2$ agent to *learn* to create knowledge of satisfied conditions in WM as sources of spread whenever the corresponding conditions are true and remove them when they is false. This not only fills the computational gap left from the Actransfer implementation, it also broadens the theory of the cognitive model of task learning.

The $PROP_2$ agent begins each instruction retrieval with an open query for any instruction, so that the result can be determined by activation. At first, the agent has no knowledge of the conditions that might be attached to that instruction. Retrieval results will therefore be random at first, since there will be no meaningful spreading and all instructions will have roughly equal activation. After a retrieval, however, the agent can examine the conditions in whatever instruction it retrieved. The agent can then learn these conditions through practice in evaluating them. When the agent finds that any single condition is satisfied, it creates WM knowledge of this fact as a substate result, and this can be chunked into an elaboration rule. As an elaboration rule, this chunk will recreate this condition fact any time the condition is satisfied in the future, and the chunk will retract, and therefore remove the fact from WM, whenever the condition is no longer satisfied. Even when examining an instruction that does not have matching conditions, this practice gradually allows the agent to learn condition-evaluating chunks that trigger spreading activation, and eventually the agent learns enough conditions so that it always retrieves instructions with satisfied conditions.

This learning process works even if the agent starts by searching its memory for an instruction in a completely random manner. However, this approach is still inefficient. Basic experimentation showed that this approach could require the equivalent of days of model time before the agent would start to reach the level of accuracy necessary to perform the task reliably, depending on how many instructions the agent had to randomly search through in SMEM in order to find an instruction with satisfied conditions. But there is no reason the agent has to search its memory blindly when it learns to perform a task, any more than humans would be expected to perform

tasks without instruction or prompts. I gave the $PROP_2$ agent the ability to use learn its procedural knowledge of the sequence of task operations based on given declarative knowledge. The agent always begins P2 by using spreading to control which instruction it retrieves. Call this method, "free recall." If after a free recall the agent finds that it retrieved an instruction with unsatisfied conditions, it then retrieves the explicit declarative retrieval sequence from its SMEM, the same sequence as used for $PROP_1$, and uses that to explicitly retrieve the next instruction. Call this method, "cued recall." The agent only falls back on cued recall if a free recall fails to give it a usable instruction by default.

As I mentioned in chapter 3, I do not attempt to model where declarative knowledge about instructions comes from. I assume SMEM knowledge that describes conditions is provided at the same time as instructions, structured in the format shown in Figure 8.3. Still, it was uncertain at the time that I developed this approach whether or not learning P2 ought to be considered declarative or procedural in terms of cognitive modeling. The functionality that I described followed from the joing constraints of P2 in PRIMs theory and Soar's spreading activation. Experimentation and evaluation of the $PROP_2$'s behavior helped answer this theoretical questions.

In summary, I introduce a new approach by which a cognitive model can learn to use the state of WM to guide LTDM retrievals. I go on to show in section 8.4 that this learning process replicates aspects of human behavior that previous models could not.

8.2 Problem 2: Gradual Chunking

The $PROP_1$ approach simulates gradual learning behavior as desired, but its approach does not fit into the broader cognitive model and does not reflect Soar learning theory. Procedural learning should be an automatic process controlled by the architecture, not deliberate agent reasoning. The results from $PROP_1$ experimentation indicated that such a mechanism was both possible and desirable. I determined that an architectural change was needed for Soar to properly support gradual procedural learning.

The work here was fairly straightforward. In line with the $PROP_1$ approach, I added gradual procedural learning to Soar, not by delaying when a learned rule would be used in practice, but by delaying when Soar would actually learn the rule. I modified the architecture so that a specific chunk must be (re)learned multiple times before the architecture actually saves it to procedural memory, where it then is able to match and fire whenever its conditions are satisfied. I again use θ_p as a parameter to control how many times the agent must attempt to learn a chunk before the chunk is saved to long-term procedural memory. The behavior of this parameter is the same as I described for $PROP_1$.

With this learning mechanism available, $PROP_2$ no longer needed agent decisions to simulate

gradual composition, and this allowed me to remove much of the overhead decision making from the $PROP_1$ design. The agent did not need to track how many times it practiced any operation, but would learn new rules simply by repeatedly practicing its task instructions. This reduced the number of computation cycles needed to process task instructions, even cycles that were not directly part of the simulated architectural processing that I omitted from $PROP_1$ results. The overall decision making and operator composition of $PROP_2$ were still inherited from $PROP_1$, however, and thus agent decision making was still used to control the binary structure of the hierarchical composition of practiced procedures. These still need to be removed from model results during evaluation.

8.3 Connecting Three-Phase Theory

During this research, I discovered a correspondence between these two types of learning and the types of learning described in Fitts and Posner's (1967) three-phase theory.

Recall that in three-phase theory the *cognitive phase* is the phase when a subject learns how to properly approach the task in practice, and learns the correct order and nature of the operations it needs to execute. Once the subject has learned what to practice and proceeds to practice it and gain expertise in that practice, it has reached the second *associative phase*. Then once the subject has gained enough expertise in that practice to be able to execute the task without significant cognitive effort or attention, and can think about other things while performing the task, it has reached the *autonomous phase*.

$PROP_2$ progresses through the classic three phases of skill learning, and particularly adds the *cognitive phase* to Actransfer computation by learning P2. When starting the first *cognitive phase*, the agent knows how to retrieve and execute instructions but does not know when specific instructions should be retrieved and practiced. Until it learns when to retrieve and practice each of the different relevant instructions for the task, retrievals are random and the agent can retrieve irrelevant instructions. But after some practice, the agent gains enough experience to always retrieve instructions that are immediately relevant, and it spends more of its time increasing its expertise by practicing those specific instructions. It is then 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 the need to use task instructions or reason over condition and action lines. The agent is then in the *autonomous phase*.

I can enable or disable the processing that corresponds to cognitive and autonomous learning, and I use this to align computational theory with the cognitive theory for human performance profiles. I can toggle P2 learning on and off by toggling whether the agent attempts an activation-based retrieval vs. whether it only follows the rote retrieval sequence the way the $PROP_1$ agent did.

And as with PROP$_1$, I can toggle whether the agent learns `Auto` rules by enabling or disabling whether it chunks the processing of its outer substate. These two settings give me two parameters for PROP$_2$, which correspond closely to the descriptions of the *cognitive phase* and the *autonomous phase*:

1. **`Learned / Known`** - whether the agent learns how and when to cognitively arrange procedures for the task or whether this knowledge is already known
2. **`Deliberate / Auto`** - whether the agent is always deliberate with cognitive control over task operations or whether it can learn automatic task execution

If P2 is `Learned`, then during execution the PROP$_2$ agent learns rules that induce spreading activation for satisfied condition lines. If P2 is `Known`, the agent *always* retrieves through cognitive control by following a declarative retrieval sequence.

The `Deliberate / Auto` parameter is the same as I used when experimenting with PROP$_1$. If set to `Auto`, the agent eventually learns a single rule (the top-most composition depicted in Figure 4.4) that performs task operations without declarative instruction or cognitive control over them. `Auto` rules effectively bypasses the entire PRIMs learning pipeline once learned. If set to `Deliberate`, this final composition step never occurs, so that execution must *always* be deliberate using the PRIMs pipeline.

A PROP$_2$ agent with `Learned` enabled has to learn P2 retrieval, which is the process of learning which skills to use and their order when executing the task. Thus, toggling `Learned / Known` effectively should toggle whether the agent needs to progress through the cognitive phase or whether it begins the task already at the associative phase. Toggling `Deliberate / Auto` toggles whether the agent gains enough task-specific specialized expertise so that it does not need to use cognition with declarative knowledge of its actions in order to perform its actions during the task. An agent that has fully learned all `Auto` rules for its task acts effectively by reflex as one procedural rule fires one after the other.

Actransfer is an (`Auto, Known`) model, since it assumes P2 retrieval functionality (by hard-coding activation behavior) and allows `Auto` task rules to be fully learned. Theoretically, the PROP$_2$ model is similar to Actransfer when using `Auto-Known`.

In summary, I identify how specific learning processes in my implementation of PRIMs theory correspond with with the phases of the three-phase theory of human learning.

8.4 Evaluation

I evaluated PROP$_2$ behavior against human behavior using the mental arithmetic task (Elio, 1986) as I did before with PROP$_1$, and I also introduced the text editors task (Singley & Anderson, 1985)

to the suite of PROPs experiments. While the arithmetic task is key for evaluating the learning curve, the editors task allows me to examine transfer in greater detail.

There are two contributions of PROP_2 that I aimed to evaluate.

1. My approach for learning P2 instruction retrieval

2. My implementation of gradual chunking

I evaluate the first by comparing PROP_2 `Known` and PROP_2 `Learned` performance in the manner I described above. This also lets me examine the theoretical correspondence between this learning behavior and *cognitive phase* learning behavior. The profile of human *cognitive phase* learning is not precise enough for me to make an exact comparison, but I can compare the difference in PROP_2 with and without `Learned` P2 and thus measure the amount of latency that this learning process adds to the model. As with PROP_1, I also compare PROP_2 `Deliberate` and PROP_2 `Auto` performance. With the theoretical foundation of three-phase theory, I can now similarly compare this learning behavior to what is expected for the *autonomous phase* in human learning. I can predict that using `Learned` should cause the agent to start off slower in its task until it learns P2, since until then it wastes time searching through LTDM for an instruction to use. I also expect similar effects for `Auto` as with PROP_1, namely that it allows the agent to converge to super-human performance.

I evaluate the second by comparing the overall PROP_2 learning curve to that of PROP_1 and Actransfer to see whether and how the PROP_2 agent reproduces gradual learning. Actransfer's gradual learning comes from ACT-R's production compilation mechanism. PROP_1's gradual learning comes from a simulation of ACT-R's production compilation within Soar. PROP_2's gradual learning reflects a union of what is required for PRIMs theory with Soar's built-in chunking approach. I hypothesized that the PROP_2 approach to gradual learning should be equivalent for practical purposes to PROP_1. The high-level theory of gradual chunking is the same, but the fact that the agent no longer uses deliberate reasoning to group and prefer pairs of instruction lines means that the PROP_2 agent could potentially generate a slightly different learning hierarchy. It is difficult to predict the exact effects, however, because of the complex cascading nature of hierarchical learning and transfer, where later iterations of learning depend on what is learned in previous iterations.

I also predict that the PROP_2 agent will be slightly faster than PROP_1, since the introduction of gradual chunking to Soar let me remove some processing from the PROP_1 design.

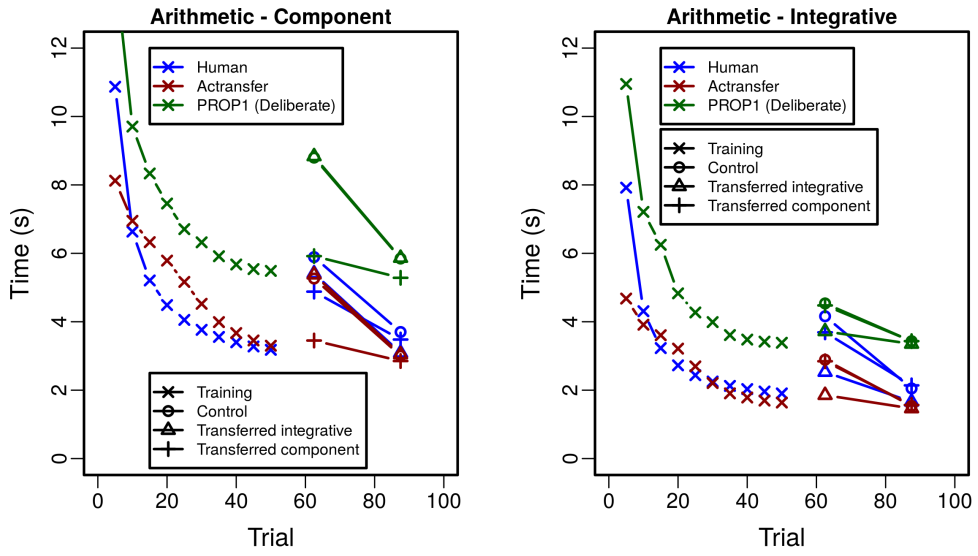
According to my methodology, I use the same task instructions used for the Actransfer agents, as well as the same latency values for visual/motor environment interaction. As with PROP_1, I also still define P2 retrieval latency for PROP_2 by copying the values used for Actransfer, since PROP_2 still does not address P6 and the way that latency is calculated. To select θ_p, I performed a parameter sweep (not shown) to find the value that provides the closest fit to the Actransfer model

for comparison. I use a threshold of 48 for the editors task models and 10 for arithmetic task models below. The Actransfer model similarly uses different learning rates for each task. Recall that PROP_1 used $\theta_p = 16$ for the arithmetic task. Due to the transition from PROP_1's explicit agent control over its learning to PROP_2's automatic architectural learning, there are slight differences in how the agents select pairs of rules to combine together. This means the value of θ_p for PROP_2 is not fully comparable to the value of θ_p for PROP_1.

As before, results for the arithmetic task are averaged over 8 repetitions. Results in the editors task for Actransfer and for PROP_2 are averaged over 12 repetitions. These models are fairly deterministic, and variance is low.

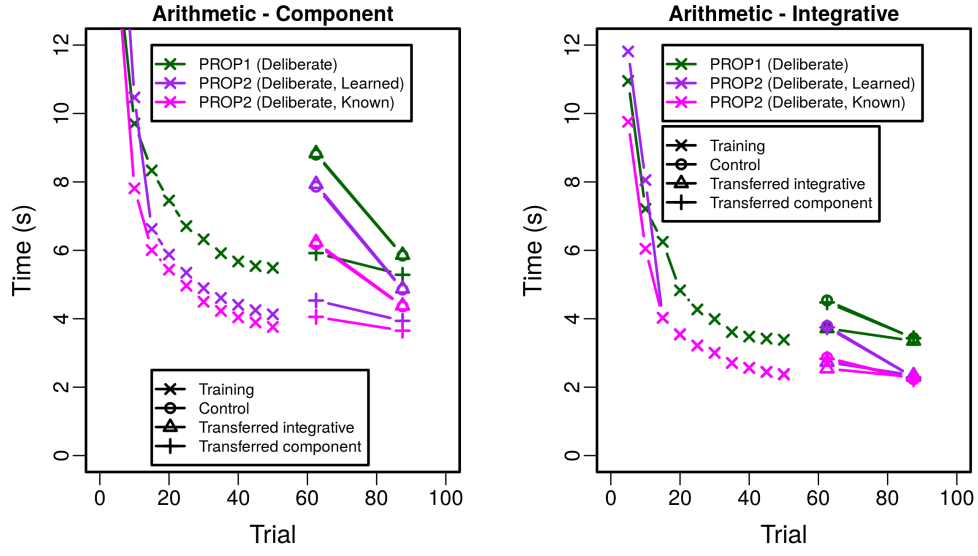
Again, I use the same timing approach as the Actransfer agent. The arithmetic model times are as discussed previously in section 7.7. The Actransfer editors model used 0.25 sec for the time to press one key as well as for the visual time to move attention from one item to another.

8.4.1 Arithmetic Task



(a) Component steps

(b) Integrative steps



(c) Component steps

(d) Integrative steps

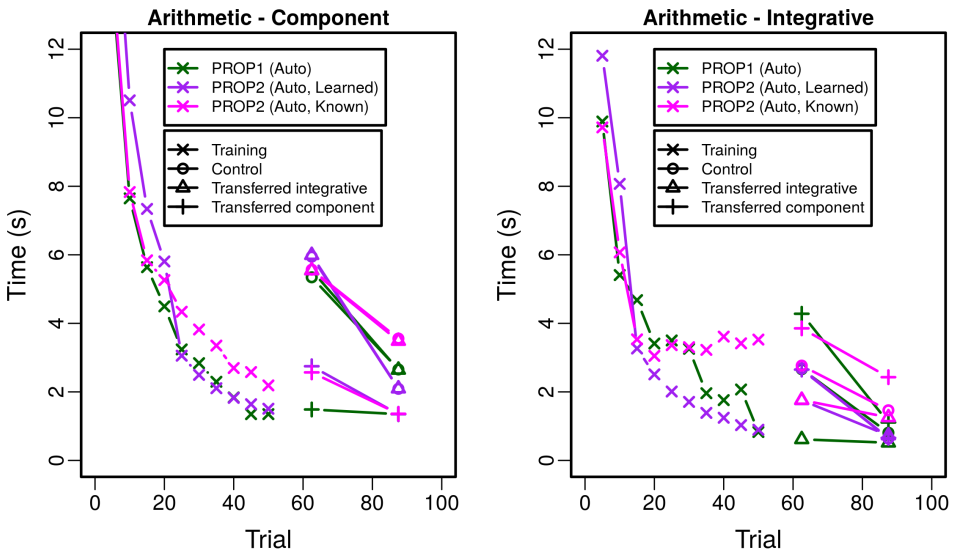
Figure 8.4: A review of human, Actransfer, and PROP_1 `Deliberate` performance alongside PROP_2 `Deliberate` performance for the arithmetic task.

I can compare PROP_2 results in the arithmetic task with those from PROP_1 to see whether and how the new gradual learning mechanism in Soar is equivalent to PROP_1's simulation of Actransfer.

Figure 8.4a and Figure 8.4b show again the results from section 7.7 for humans, the Actransfer model, and the PROP_1 (`Deliberate`) model in the arithmetic task. Figure 8.4c and Figure 8.4d

show the PROP$_2$ (`Learned, Deliberate`) and (`Known, Deliberate`) models.

One immediately visible result is that PROP$_2$ performance is faster than PROP$_1$ by a fairly constant amount. This is as predicted, due to the way the gradual chunking mechanism in Soar simplified agent processing. Also worth noting is that the power-law performance from PROP$_1$ PRIM resolution is preserved in PROP$_2$. One can also see that the `Learned` model catches up to the `Known` model after 15 trials. It does not catch up entirely in Figure 8.4c due to the fuzzy nature of using activation. For component steps, some instructions were similar enough to each other in their conditions that the agent could sometimes retrieve the wrong instruction and delay its performance.



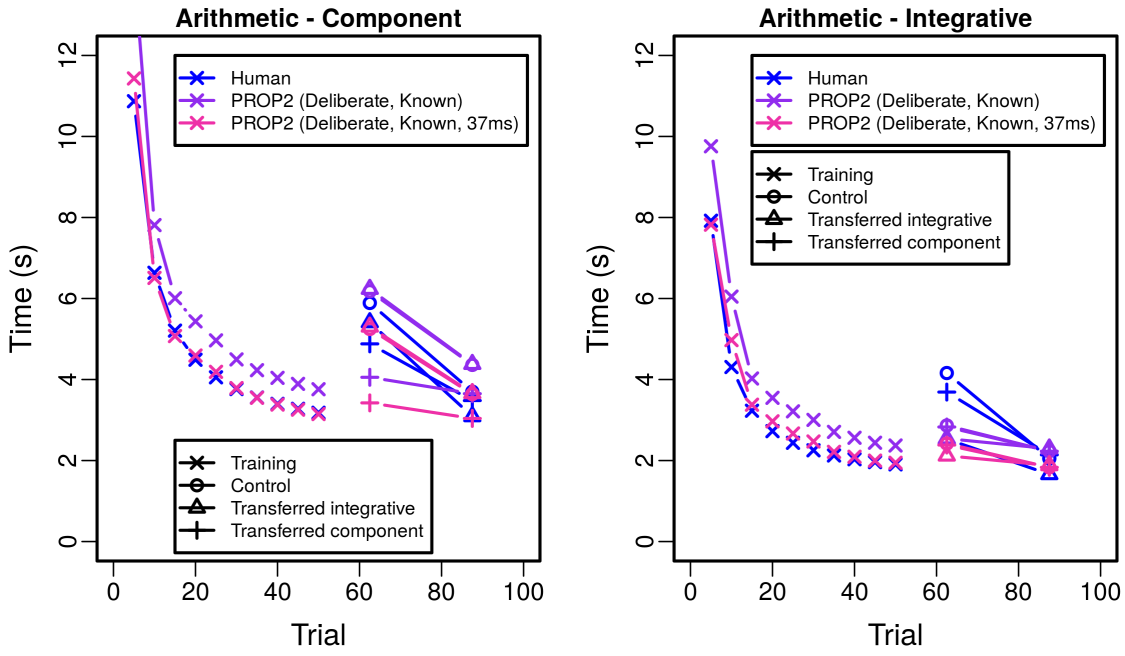
(a) Component steps

(b) Integrative steps

Figure 8.5: PROP$_2$ `Auto` performance for the arithmetic task.

Figure 8.5 shows the `Auto` models. As with PROP$_1$, the `Auto` parameter results in eventual super-human performance. The (`Auto, Known`) learning curve is not smooth. In the integrative steps, the `Auto` parameter even results in worsening performance after about 20 trials of training. This same effect was observed with PROP$_1$, and occurs when `Auto` rules perform the task automatically without updating the agent's declarative retrieval sequence. The agent must then spend time to step through its memory of the task steps to catch up to the task state. By contrast, the (`Auto, Learned`) agent does not exhibit this catastrophic memory failure, and proceeds with smooth monotonic learning after trial 20. This indicates that the PROP$_2$ learning approach for conditions and spreading activation is robust against disruption from `Auto` rules.

I also observe that while there are many summed components in the total model latency, such as retrieval time and motor latency, the amount by which (`Deliberate, Known`) behavior differs



(a) Component steps

(b) Integrative steps

Figure 8.6: `Deliberate-Known` PROP$_2$ model with 50 msec and 37 msec for decision latency.

from human performance is nearly an exact multiple of decision cycle latency. Setting decision cycle time to 37 msec results in almost an exact fit to the human curve, as shown in Figure 8.6.[2] While not a standard timing, this is consistent with neural modeling that predicts cycle times of approximately 40 msec for simple actions (Stewart et al., 2010).

Table 8.1 shows the goodness-of-fit measures for the various PROP$_2$ models for the component and integrative operations, respectively. Examined alongside the various plots, the differences in r^2 above 0.98 appear to be fairly negligible. One can see that the the PROP$_2$ (`Deliberate, Known, 37ms`) model has the best measures, as is expected from viewing Figure 8.6, with the exception of r^2 for integrative steps. In that instance, the relatively slow performance on trial 10 gives (`Deliberate, Known, 37ms`) a slightly worse fit than (`Deliberate, Known`). The second closest model in terms of error scores is the Actransfer model, though the PROP$_2$ (`Deliberate, Known`) model is similar.

[2]This can not be achieved by changing θ_p, as that alters the learning curve shape in addition to scale.

	Component Steps			Integrative Steps		
Model	r^2	*MAE*	*MAPE*	r^2	*MAE*	*MAPE*
Actransfer	0.808	0.84	15.7	0.758	0.58	14.2
$PROP_1$ (Deliberate, Known)	0.994	2.71	61.3	0.969	2.05	72.2
$PROP_2$ (Deliberate, Learned)	0.993	2.00	36.0	0.960	1.29	36.0
$PROP_2$ (Deliberate, Known)	0.995	1.04	20.2	**0.991**	0.88	28.6
$PROP_2$ (Auto, Learned)	0.976	2.28	44.0	0.945	1.23	36.1
$PROP_2$ (Auto, Known)	0.992	0.89	16.6	0.938	1.19	45.9
$PROP_2$ (Deliberate, Known, 37ms)	**0.997**	**0.12**	**1.84**	0.988	**0.18**	**6.05**

Table 8.1: $PROP_2$ goodness-of-fit measures for arithmetic data. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

Component Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		0.48	1.01
Actransfer		-0.11	1.81
$PROP_1$ (Deliberate, Known)		-0.04	2.88
$PROP_2$ (Deliberate, Known)		-0.06	2.12
$PROP_2$ (Deliberate, Learned)		-0.07	3.33
$PROP_2$ (Auto, Known)		0.02	3.00
$PROP_2$ (Auto, Learned)		-0.04	3.20
	Percent Transfer		
Human	11.75%	27.38%	44.63%
Actransfer	38.09% (+26.34)	34.51% (+7.13)	95.23% (+50.60)
$PROP_2$ (Deliberate, Learned)	26.70% (+14.95)	**25.30% (-2.08)**	92.08% (+47.44)
$PROP_1$ (Deliberate, Known)	**13.50% (+1.75)**	12.55% (-14.82)	88.72% (+44.08)
$PROP_2$ (Deliberate, Known)	38.36% (+26.62)	36.96% (+9.58)	92.50% (+47.86)
$PROP_2$ (Auto, Known)	36.36% (+24.61)	36.77% (+9.40)	92.84% (+48.21)
$PROP_2$ (Auto, Learned)	39.89% (+28.14)	39.30% (+11.93)	**83.21% (+38.58)**
$PROP_2$ (Deliberate, Known, 37ms)	34.59% (+22.84)	32.90% (+5.52)	91.42% (+46.79%)

Table 8.2: $PROP_2$ transfer for arithmetic component steps.

Integrative Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		1.63	0.47
Actransfer		1.03	0.04
$PROP_1$ (Deliberate, Known)		0.82	0.06
$PROP_2$ (Deliberate, Known)		0.32	0.04
$PROP_2$ (Deliberate, Learned)		1.05	0.04
$PROP_2$ (Auto, Known)		1.00	-1.09
$PROP_2$ (Auto, Learned)		0.92	0.03
	Percent Transfer		
Human	-1.62%	71.84%	19.56%
Actransfer	**36.89% (+38.51)**	89.07% (+17.23)	**39.14% (+19.57)**
$PROP_1$ (Deliberate, Known)	65.34% (+66.96)	89.95% (+18.11)	67.10% (+47.54)
$PROP_2$ (Deliberate, Known)	83.25% (+84.87)	94.21% (+22.37)	84.50% (+64.94)
$PROP_2$ (Deliberate, Learned)	62.54% (+64.16)	90.69% (+18.85)	63.57% (+44.01)
$PROP_2$ (Auto, Known)	146.90% (+148.52)	208.84% (+137.00)	79.79% (+60.23)
$PROP_2$ (Auto, Learned)	61.53% (+63.15)	**81.45% (+9.61)**	62.17% (+42.61%)
$PROP_2$ (Deliberate, Known, 37ms)	80.37% (+81.99)	92.82% (+20.98)	81.89% (+62.33)

Table 8.3: $PROP_2$ transfer for arithmetic integrative steps

Table 8.2 and Table 8.3 show the transfer for $PROP_2$ alongside Actransfer and $PROP_1$. There is no clear trend for which model provides the best fit. It is worth noting that $PROP_2$ (`Deliberate, Known, 37ms`) does not have a particularly close fit in percent transfer except in component step transfer for integrative and component tests. Additionally, Actransfer has some of the worst transfer fits for component steps, but two of the best fits for integrative steps. The lack of a clear trend implies that something is missing in the model of transfer for this task. Later experimentation with $PROP_3$ reveals some insights into what this might be, as I discuss in section 9.5.

8.4.2 Editors Task

Figure 8.7 shows human and Actransfer editors data alongside $PROP_2$ results for the two parameters. The top two plots show human and Actransfer performance at different scales. The lower two plots show $PROP_2$ performance using the different learning parameters. $PROP_2$ is fairly deterministic even when using `Learned`. Standard error is < 0.02 sec for each point for `Learned` models and < 0.008 sec for `Deliberate` models.

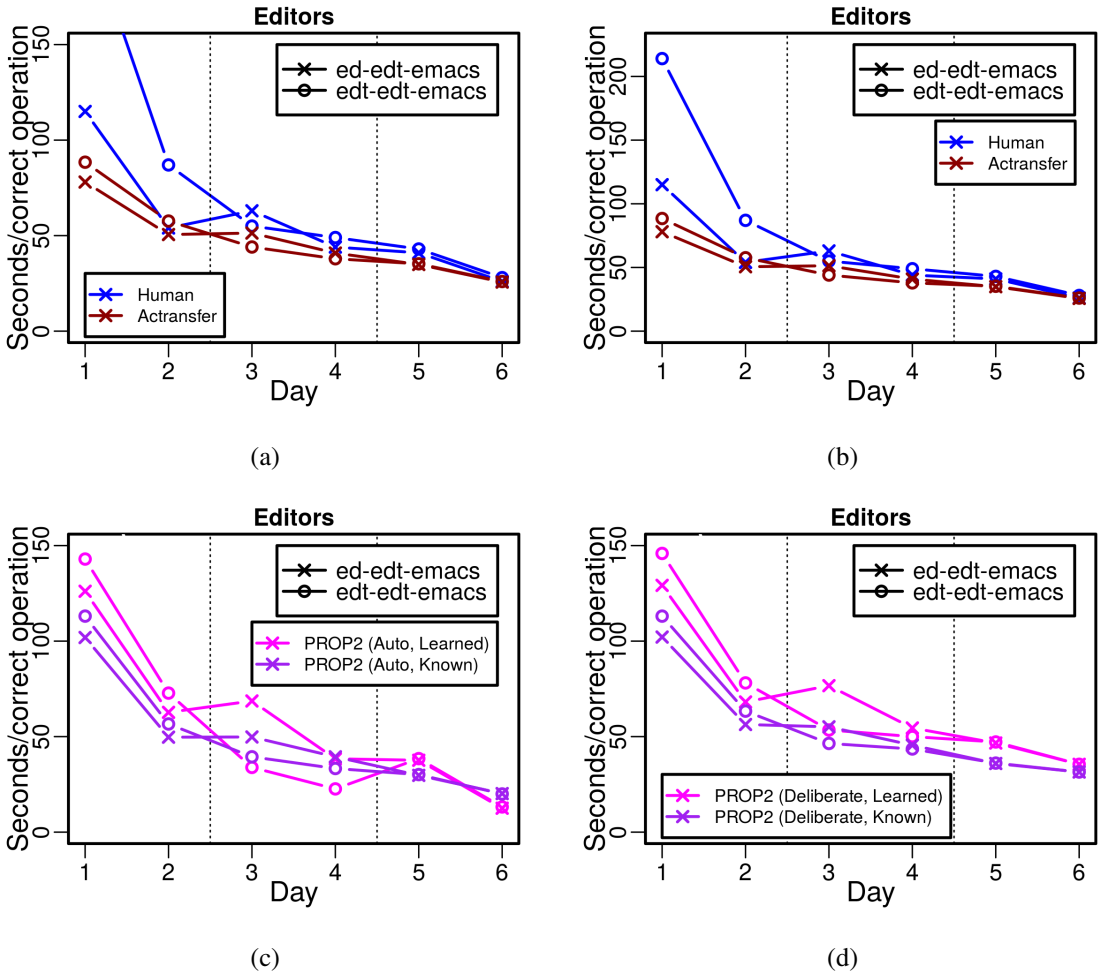
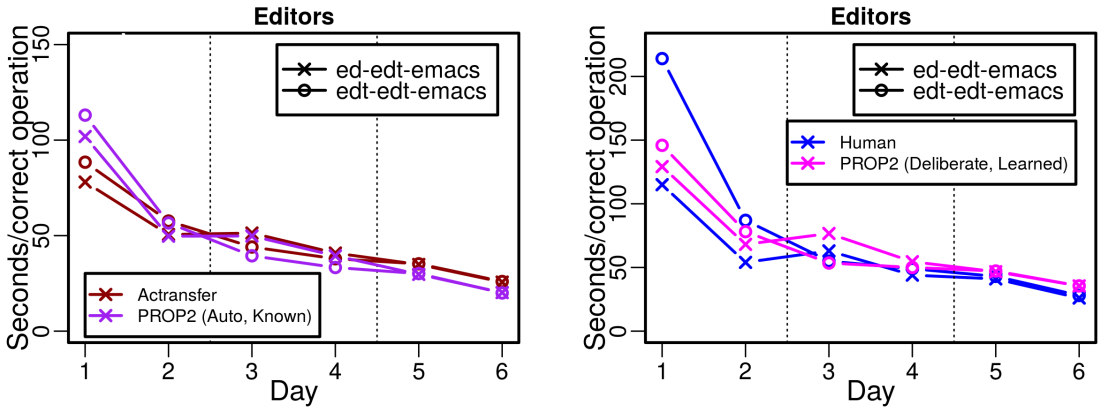


Figure 8.7: Human, Actransfer, and PROP$_2$ model results for the editors task.

One immediately noticeable result is that `Learned` performance in PROP$_2$ is always slower on days 1-2, which provides a closer fit to human behavior (as I discuss further below). As predicted, the agent is slower on these days because it spends extra time searching LTDM for instructions with satisfied conditions until it learns P2. After the agent learns P2 by around days 3-4, however, `Learned` models are not far behind the `Known` models in performance. This demonstrates that `Learned` models learn P2 in parallel with normal PRIMs procedures for executing the task. `Deliberate, Learned` performance does remain slightly slower than `Deliberate, Known` performance throughout all six days. This is because the `Known` model still experiences some failed retrievals on occasion even after learning condition satisfaction.

As predicted, `Auto` models achieve super-human performance by day 6, at under 20 sec. Human and Actransfer models, by contrast, end near 30 sec, as do PROP$_2$ `Deliberate` models. As with PROP$_1$, learned `Auto` rules allow the agent to skip the PRIMs instruction process entirely, and this leads to much faster performance.

I also observe that, as desired, these PROP$_2$ models provide gradual learning behavior comparable to Actransfer's using the gradual learning mechanism that I introduced to Soar.



(a) (b)

Figure 8.8: Best PROP$_2$ model fits to human and Actransfer data for the editors task.

Figure 8.8 shows the best PROP$_2$ fits for the Actransfer and human editors plots, respectively. As stated earlier, the Actransfer model is (`Auto, Known`) in theory, and in this case the PROP$_2$ (`Auto, Known`) model is a close fit to Actransfer. But as with PROP$_1$, the PROP$_2$ agent performs faster after learning due to Soar's more absolute use of learned `Auto` rules, and it is slightly slower at the start due to its PRIM resolution. By contrast, the PROP$_2$ (`Deliberate, Learned`) model is the closest fit to human data in terms of shape. The cost of using `Learned` accounts for slower human performance during the first days, while its continued `Deliberate` learning paces its progress to finish at nearly the same performance as humans on day 6. Performance on days 1-3 is slightly fast for EDT but slightly slow for ED, implying the task instructions for EDT should be more complex and those for ED slightly less so.

I do not show the model with decision cycle time set to 37 msec as I did with the arithmetic task. For the editors task, this reduces the scale of agent timing by < 3 sec, just as it does with the arithmetic task. This is negligible in the scale of the editors task, where declarative retrieval time dominates.

Table 8.4 shows the r^2 and other goodness-of-fit measures for the various editors models. The PROP$_2$ (`Deliberate, Learned`) model has the best overall fit in terms of shape, as indicated by the high r^2 scores, and also has the least error for the EDT-EDT-EMACS learning condition. The PROP$_2$ (`Deliberate, Known`) model has the least error for the ED-EDT-EMACS condition, however, because of its slightly faster performance.

Table 8.5 shows transfer scores for the editors task models. The formula for measuring percent

Model	**ED-EDT-EMACS**			**EDT-EDT-EMACS**		
	r^2	*MAE*	*MAPE*	r^2	*MAE*	*MAPE*
Actransfer	0.965	10.27	13.4	0.958	31.13	26.7
PROP_2 (Deliberate, Learned)	**0.994**	11.28	22.4	**0.995**	**15.21**	**13.9**
PROP_2 (Deliberate, Known)	0.977	**5.86**	**10.8**	0.991	24.83	21.6
PROP_2 (Auto, Learned)	0.980	7.99	17.9	0.962	25.35	34.2
PROP_2 (Auto, Known)	0.986	8.71	16.8	0.993	30.57	33.5

Table 8.4: PROP_2 goodness-of-fit measures for the editors task. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

transfer comes from Singley and Anderson (1987) for the original experiment, and is as follows:

$$\%transfer = \frac{(single_1 - transfer_n)}{(single_1 - single_n)} \times 100$$

In this formula, $single_1$ and $single_n$ are the subject performance times when practicing a single editor on day 1 and day n. The value of $transfer_n$ is the subject performance on day n for the same editor after practicing a *different* editor for all preceding days. For example, compare two agents, "ED-EDT" who practiced ED for days 1-2 followed by EDT on day 3, and "EDT-EDT" who practiced EDT all three days. If "EDT-EDT"'s average performance was 100 sec on day 1 and 50 sec on day 3, while "ED-EDT"'s was 60 sec on day 3, then the transfer from ED to EDT would be calculated as $(100 - 60)/(100 - 50) \times 100 = 80\%$.

The original report on human transfer distinguished between time spent planning to move a cursor to a line and time spent planning to actually edit text. I similarly report these as well as the overall "Global" transfer in Table 8.5. Also in the same manner as the original report, I average together transfer from ED and EDT to EMACS, as the first two are single-line editors and relatively similar to each other, while EMACS is a visual editor and fairly different from the other two. As before for reporting arithmetic transfer, I note the differences between model and human transfer in parentheses to the right of each model transfer score. Models with the closest transfer to humans are marked in bold.

Overall, the PROP_2 (`Deliberate, Known`) model exhibits transfer most similar to humans. Actransfer has the closest transfer score when averaging together EDT and ED transfer toward EMACS, however. Though the (`Deliberate, Learned`) model has the more human-like learning curve shape, its curve is shifted slightly higher than the other models for days 2-6, particularly on day 3, which is when transfer is measured.

Most models exhibit less transfer between the two line editors than humans overall. The PROP_2 (`Auto, Known`) agent in particular exhibits instances of negative transfer. This results from the catastrophic forgetting behavior that can result from that parameter combination, as I described

Model	*Global*	*Planning Move to Line*	*Planning Edit Text*
ED to EDT Editors Transfer			
Human	94.97%	87.50%	104.62%
Actransfer	83.54% (-11.42)	82.03% (-5.47)	88.35% (-16.27)
$PROP_2$ (Deliberate, Learned)	74.90% (-20.07)	48.72% (-38.78)	27.44% (-77.18)
$PROP_2$ (Deliberate, Known)	**86.95% (-8.02)**	**85.88% (-1.62)**	**88.47% (-16.15)**
$PROP_2$ (Auto, Learned)	68.12% (-26.85)	32.11% (-55.39)	14.66% (-89.96)
$PROP_2$ (Auto, Known)	85.92% (-9.05)	52.31% (-35.19)	-205.60% (-310.22)
EDT to ED Editors Transfer			
Human	97.18%	91.84%	99.09%
Actransfer	90.85% (-6.33)	89.93% (-1.91)	91.80% (-7.29)
$PROP_2$ (Deliberate, Learned)	84.65% (-12.54)	73.85% (-17.99)	34.99% (-64.10)
$PROP_2$ (Deliberate, Known)	**96.22% (-0.96)**	**92.65% (+0.81)**	**102.23% (+3.14)**
$PROP_2$ (Auto, Learned)	73.31% (-23.87)	60.75% (-31.09)	-2.46% (-101.55)
$PROP_2$ (Auto, Known)	95.82% (-2.37)	88.04% (-3.80)	197.80% (+98.71)
EDT/ED to EMACS Editors Transfer			
Human	64.81%	61.03%	62.29%
Actransfer	**63.41% (-1.40)**	**66.70% (+5.66)**	**57.26% (-5.03)**
$PROP_2$ (Deliberate, Learned)	74.47% (+9.65)	13.39% (-47.64)	16.66% (-45.64)
$PROP_2$ (Deliberate, Known)	84.07% (+19.25)	91.04% (+30.01)	126.92% (+64.63)
$PROP_2$ (Auto, Learned)	60.06% (-4.76)	42.81% (-18.22)	12.36% (-74.65)
$PROP_2$ (Auto, Known)	73.51% (+8.70)	40.77% (-20.26)	-7.39% (-69.68)

Table 8.5: $PROP_2$ transfer in the editors task, along with differences compared to humans.

with $PROP_1$ and which I further discuss in section 8.5. In this case, the agent ends up with worse performance on day 3 than on day 1 when the learned `Auto` rules interfere with the agent's ability to efficiently access declarative knowledge about the task. This causes the denominator in the transfer function to be negative.

These results support the idea that a model of this task should account for the process of learning P2. The $PROP_2$ models that do so with `Learned` provide the closest fits to human performance by accounting for the extra time required for learning during the first few days of the task.

8.5 Discussion

These results demonstrate a general principle for human modeling, that consideration for the stages of learning and these agent parameters is critical for human modeling. Whether an subject learns P2 during the experiment or has already learned it beforehand should be reflected in the model.

With $PROP_2$, I successfully imbued a PRIMs agent with the ability to learn associations between states in WM and task instructions in SMEM through experience. This allows spreading activation to bias the agent to retrieve instructions that have satisfied condition lines. I model *cog-*

nitive phase learning with a novel use of Soar chunking by which chunks create or retract sources of spreading activation. This general learning method is not limited to modeling PRIMs, but can be applied whenever an agent needs to learn context-appropriate retrievals.

The PROP$_2$ agent demonstrates progress through Fitts and Posner's (1967) three stages of skill learning: it begins in the *cognitive phase* as it `Learns` which task instructions to retrieve and perform, progresses to the *associative phase* as it then compiles hierarchical instruction lines for the instructions that it practices, and finally reaches the *autonomous phase* as it collects `Auto` rules that perform the various task operations without the need for declarative instructions.

In the editors task, `Learned` P2 accounts for initial human behavior in ways `Known` P2 could not. However, in the arithmetic task, `Known` models were most accurate. I 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*. Human subjects in the editors task, however, were not allowed to practice the task prior to measurement. Thus, results should be expected to include a cognitive phase of learning. Therefore, any accurate computational model of that task should account for cognitive phase learning, in addition to other phases.

The relation between the `Learned` setting in PROP$_2$ and the cognitive phase helps clarify the boundaries between declarative and procedural learning within a model of PRIMs theory. Cognitive phase learning is often considered specifically a type of declarative learning rather than procedural learning (Kim & Ritter, 2015). Even in PROP$_2$, the chunks that the agent creates for `Learned` mode merely create declarative associations between WM conditions and SMEM instructions. If this kind of learning does correspond to the cognitive phase, as it seems to, then it likely should not be modeled using procedural learning processes such as chunking but as a declarative learning process. In any case, the PROP$_2$ model also does not address the origin of the declarative SMEM structures that allow normalized spreading. A complete model of declarative learning is outside the practical scope of this thesis, but some mechanic for P2 is still required for PRIMs theory.

The question of *learning* P2 is therefore outside the scope of my modeling work for this thesis, and represents a much larger scope of research than was originally anticipated. This PROP$_2$ model for learning P2 helps draw this distinction and inform an understanding of how PRIMs processing is situated with respect to other learning processes.

As a topic of future investigation, it would make sense that individual humans both construct and modify the declarative structures of their task understanding during task execution in a manner similar that described in the PDL model (Gray & Lindstedt, 2017).

It should be noted that a newer experimental PRIMs architecture was independently developed

that learns associative connections from declarative goals to practiced task instructions using a reinforcement learning function (Arslan et al., 2017). This is similar to what PROP$_2$ does at a high level, though neither the details of this mechanism nor what it represents for cognitive theory have been published to my knowledge.

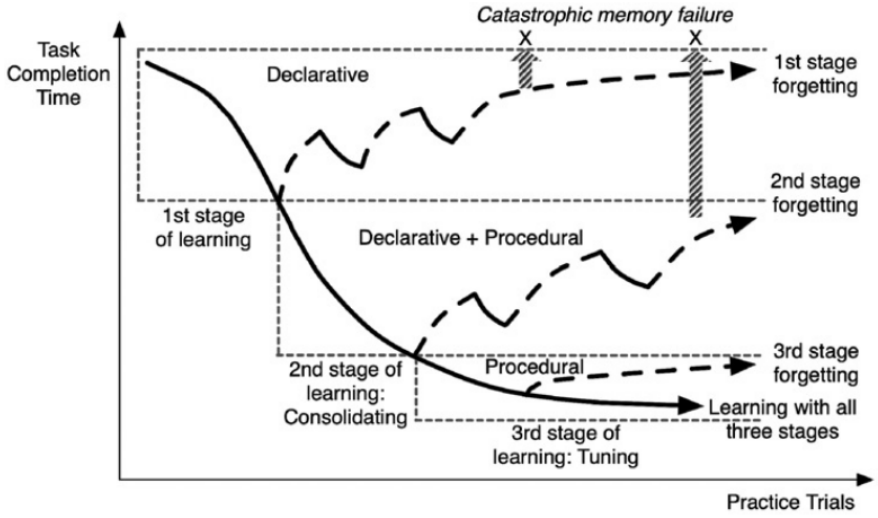


Figure 8.9: The KRK model of learning phases and catastrophic memory failure, taken from (Kim et al., 2013).

The PROP$_2$ agent also exhibits a pronounced catastrophic forgetting effect similar to that described by the KRK theory of skill learning, depicted in Figure 8.9. The PROP$_2$ agent begins to learn `Auto` rules, and then these interfere with its ability to access declarative knowledge about what to do next *if* the agent does not also have a robust set of `Learned` rules for automatically accessing that knowledge. In KRK theory, this kind of catastrophic memory failure is said to result from memory *decay*. That is, the agent is unable to quickly retrieve the declarative knowledge it needs because the activation of that knowledge is too low. The (`Auto, Known`) agent's behavior is because of extra effort used to retrieve instructions rather than because of activation decay, but the high-level theory that the declarative knowledge is less accessible remains the same.

In human performance research, it is known that autonomous behavior is observed more in overtly motor tasks, and is less possible for cognitive reasoning tasks. In PROP$_2$, though `Auto` allows faster execution, I observed that `Deliberate` achieves the closest human performance in both tasks. Within my model this similarly implies that humans do not perform these tasks entirely by reflex after training, but continue to reason over each step. The editors task requires some involved motor skills related to typing. However, the task subjects were already experienced typewriter typists, and already had mastery of autonomous typing skills. What they lacked was the cognitive reasoning skills for using these computer editors.

Taken together, experimentation with PROP$_2$ implies that the (`Deliberate, Known`) con-

figuration is most appropriate for PRIMs processing. `Auto` makes more sense for a model of motor skill learning, and `Learned` makes more sense for a model of declarative learning and reasoning. The next and final iteration of PROPs, PROP_3, was designed with these principles in mind. However, the PROP_2 use of spreading activation in combination with chunking in Soar is novel and powerful, and it might be employed in other Soar agents in the future whenever there are retrievals that need to be based on rule-like WM conditions.

Gradual learning in Soar takes the form of delaying the addition of a rule to procedural memory until its value passes a parameter threshold, θ_p. The arithmetic task used $\theta_p = 10$, while the editors task used $\theta_p = 48$. This threshold as a parameter is still task-specific, and corresponds to the learning rate parameter in Actransfer and ACT-R. This learning rate in cognitive models is considered to correspond to the amount of prior life-experience the average subject transfers into a task, where faster learning stands in for higher transfer (Anderson et al., 2019). Thus, this task-specific parameter is likely impossible to remove until I achieve a life-long learning agent that acquires actual life-experience prior to task experiments. The extent to which this parameter represents prior declarative learning or prior procedural learning is impossible to determine with certainty given the current understanding in the field, but it is likely closely tied to the question of adult PRIMs learning discussed in section 10.2, since it is a question of how adults might transfer prior experience.

The PROP_2 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 my 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 present intriguing questions for future study, but are beyond the scope of this thesis. However, as I will describe in the next section, there is another, perhaps more elegant, way to achieve a human-like model that defines instruction retrievals without the need for non-standard decision cycle times.

CHAPTER 9

PROPs Iteration 3: Defining Decision Making and Timing

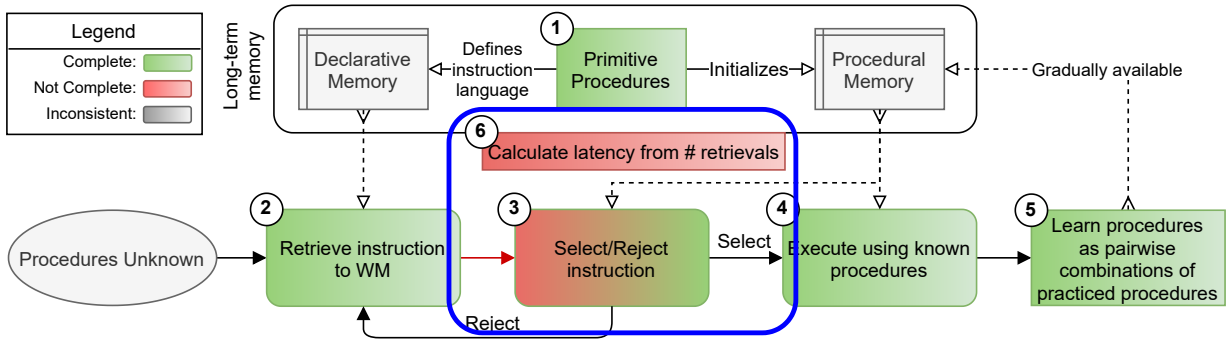


Figure 9.1: The $PROP_2$ flow diagram and its completeness for phases 1-6. $PROP_3$ addresses incomplete computation circled in blue.

I previously mentioned that there are two apparent Soar solutions for P2 retrieval selection. The first is the spreading activation approach I just described for $PROP_2$. The second uses Soar's theory for decision making to influence P2. I apply this solution in $PROP_3$.

$PROP_3$ fills the last gaps in the PRIMs model that are needed to satisfy my desiderata. These are circled in blue in Figure 9.1. While $PROP_2$ did address the P2 gap, $PROP_3$ also addresses the P3 problem, in which the Actransfer and PROP models up to now have not had the ability to use architectural processes for decision making. This problem was because P2 only ever retrieved a single instruction at a time, leaving only one choice for P3 to either accept or reject. I also address P6 by removing task-specific parameters from the model timing. I specifically remove $T_{retrieve}$ from the model timing and require the model timing to be purely a function of the agent's processing cycles. In so doing, we distinguish a new source of processing latency that was not represented in prior models. Thus, $PROP_3$ specifically targeted the following two problems:

1. The Actransfer model's instruction retrieval (P2) does not provide decision making (P3) with more than one choice of task response at a time.

2. The Actransfer model uses task-specific scaling to tune latency (P6) for specific task results.

I explain each of these in turn in the coming sections. With these changes, $PROP_3$ unifies PRIMs theory and Soar theory for all of P1-P6 in a manner that satisfies my desiderata for this thesis work.

9.1 Problem 1: Choice-based Decision Making

Actransfer, $PROP_1$, and $PROP_2$ models are not able to use decision making to select an instruction from among multiple concurrent choices. But this ability is necessary for P3 to satisfy D4 as a consistent architectural model of human processing. In order to allow P3 decision making to select from among multiple instruction choices, multiple instructions must be in WM at once. However, the process of sequentially retrieving multiple instructions to WM before making a choice among them could take prohibitively long for many tasks, for the reasons discussed in sections 4.2.2 and 8.1.1.

These constraints imply that the agent must retrieve multiple task instructions into WM at once, in parallel. If P2 retrieves a *set* of instructions at once, and *if* these are the set of instructions that have satisfied conditions and thus are relevant to the agent's P3 decision making, then P3 could process them as a batch and select one instruction for execution. This would satisfy both PRIMs theory and architectural decision making. The problem then is how could P2 determine which *set* of instructions to retrieve at once? One might consider having the agent retrieve a cluster of the most highly-activated instructions from LTDM at once. However, neither Soar nor ACT-R support such a retrieval. I identify another solution, however, that is provided by Soar PSCM theory.

In $PROP_3$, I organize task instructions in LTDM according to the agent's *problem spaces*, that is the spaces of operators relevant to each known goal or subgoal. I modified the agent so that during a single P2 cycle it selects and retrieves a bundle of instructions associated with a single problem space.

Not all of the instructions for the operators of a single problem space will necessarily have satisfied conditions at once, just as not every operator for a problem space will be proposed at once. But by definition, each instruction is relevant for the agent's problem solving goals. The problem space provides a way that the agent can reduce the scope of instructions it considers to a relatively small yet contextually-relevant size.

Soar problem spaces are hierarchical, as described in section 6.4. The agent's operator choices in one problem space can represent choices for other problem spaces that the agent can enter. As I explain, this means that organizing instructions by problem space unifies the solutions for both P2 and P3. In P3, the agent's decision making selects which problem space the agent enters next, and

this determines the bundle of instructions the agent retrieves next via P2. The instructions that the agent retrieves with P2 then define the choices for future problem spaces that the agent can select with P3, and so on.

While developing this approach, I discovered a close similarity between the computation of Soar problem spaces for PRIMs theory and that of task sets from psychology research. At the same time, I also observed that this approach allows a task-general feature set for Soar's Reinforcement Learning (RL), by which the $PROP_3$ agent can *learn* P3 for task decision making. I explore both of these subjects briefly in this thesis and use them in the $PROP_3$ models that I evaluate in section 9.4, but both represent intriguing avenues for further future study.

In the rest of this section, I first explain my solution for P2/P3 in more detail. I then discuss the similarity between the $PROP_3$ approach and and task set theory, followed by an explanation of how I apply task-independent RL with this same approach.

9.1.1 Procedure Contexts

I define a **procedure context** as a declarative structure that includes the instructions for a problem space of operators in Soar. A single procedure context does not correspond to a single condition/action rule, as with an Actransfer instruction, but to a *set* of *operators* that can be proposed within the same problem space. This could include instructions for proposal, preference, and/or apply rules, depending on the nature of the problem space.

To support different kinds of problem spaces, I define two types of procedure context, which also correspond to Soar's distinction between operator proposal rules and operator apply rules: For problem spaces of different competing operators that represent different task behaviors, an **elaboration context** instructs conditions and actions for elaborations and proposal rules in Soar. For problem spaces in which an agent needs to apply a single action for an operator in a parent problem space, an **apply context** instructs actions for a Soar apply rule.

The distinction between Soar operator proposals and Soar operator applications aligns with the distinction between P3 and P4 in PRIMs theory. I use elaboration contexts to represent PRIMs condition lines, and apply contexts to represent PRIMs action lines. Thus, I separate condition lines and action lines into different structures in LTDM. I arrange procedure contexts hierarchically in Soar's SMEM according to the hierarchy of operator proposals and applications for a task, and this hierarchical structure *collectively* forms the PRIMs instructions for that task.

Figure 9.2 depicts the processing of $PROP_3$ using procedure contexts in Soar's memory systems. The layout is the same as that of Figure 4.3 on page 28, but for Soar and $PROP_3$ instead of Actransfer. The figure shows procedure contexts as triangles, and each represents a single declarative memory structure that describes the set of operators for a particular problem space of the

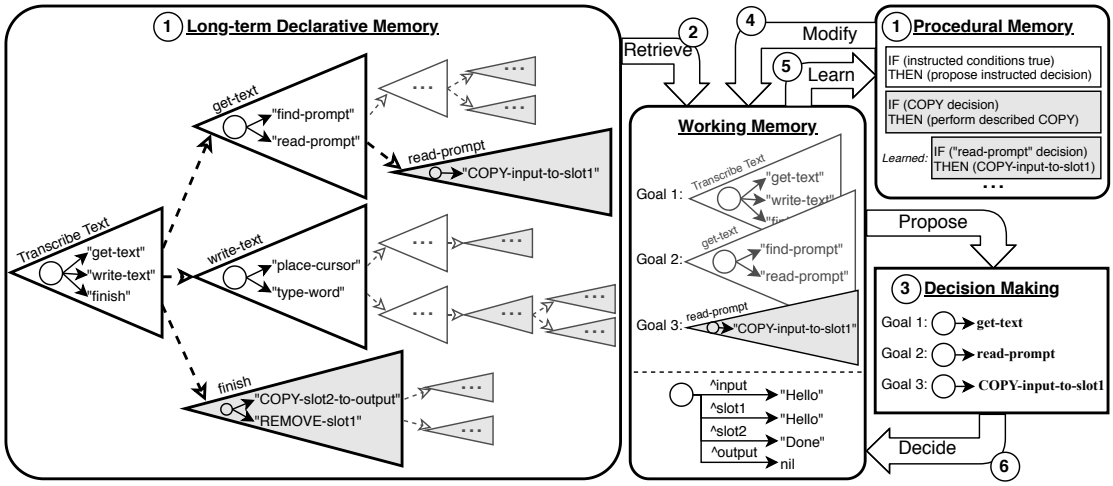


Figure 9.2: The $PROP_3$ procedure context model in Soar. Procedure contexts and rules for condition lines are shown in white, and for action lines in gray. Circled numbers mark corresponding PRIMs phases.

agent's task. These are arranged hierarchically in LTDM. All PRIMs instructions for any task are encoded within such a procedure context hierarchy in $PROP_3$. The procedure contexts in this figure are for the transcribe-text task, described previously in chapter 4. Again, the goal of this task is to copy a prompted line of text into a computer text editor, and this task is broken into component problem spaces such as "get-text" and "write-text."

In the figure, the agent has six elaboration contexts (white) in LTDM that represent its understanding of the hierarchical goal structure for the task. For the $PROP_3$ agent to begin operation, I give it the command to perform the transcribe-text task, and the task name becomes a keyword that allows the agent to directly retrieve the root "Transcribe Text" procedure context into its WM. The name of the procedure context represents the agent's goal for the corresponding state or substate the procedure context is retrieved into. In this case, the agent's top-level goal, marked "Goal 1" in the figure, is "Transcribe Text." The "Transcribe Text" procedure context is an elaboration context that includes the condition lines for three different operators, which in Actransfer would be three different PRIMs instructions, "get-text," "write-text," and "finish." In this example, assume that whenever the agent does not know what text it should be transcribing, this satisfies the conditions for "get-text." When it does know what text to write, this satisfies the conditions for "write-text." In $PROP_3$, the agent proposes any instructed operator as soon as its condition lines are satisfied in WM. For this example, assume only one of these operator is proposed at a time, but in $PROP_3$ any number of operators can be proposed in parallel so long as their condition lines are satisfied, just as Soar theory prescribes.

The first task action needed for the "Transcribe Text" task is to read the text prompt. Assume the agent first proposes and selects the "get-text" operator. There is no single apply rule that can

apply this operator because its goal requires an extended sequence of actions over time (find the prompt if needed, read the prompt once it is found). Since the agent does not know an apply rule for the "get-text" operator, an impasse arises and the architecture creates a new subgoal and WM substate. The new subgoal and corresponding substate are marked "Goal 2" in the figure. Because the agent knows the goal of "Goal 2" is to apply "get-text," it retrieves the "get-text" procedure context into the substate from SMEM without any search. In this way, I use the Soar model of navigating problem spaces to define processing for P2/P3 in PRIMs and solve the problem of accessing PRIMs instructions from LTDM.

If the agent in this example is not yet looking at the text prompt, then it can similarly select the "find-prompt" operator and enter another subgoal to carry out that action. However, let's assume the agent is already looking at the prompt, and instead proposes "read-prompt." As before, an impasse arises because the agent does not have an apply rule for "read-prompt," and therefore it makes "read-prompt" its subgoal, marked "Goal 3" in the figure. However, when the agent then retrieves the "read-prompt" procedure context, it finds that this is not an elaboration context like "Transcribe Text" and "get-text," but is an apply context. This apply context instructs a single action line, "COPY-input-to-slot1." As soon as the agent retrieves this apply context into WM for "Goal 3," a primitive apply rules fires and performs the described COPY operation. Now the "read-prompt" operation is complete, and the agent can exit that subgoal. In fact, the "get-text" operation is now also complete, and the agent exits that subgoal and proceeds to propose and apply "write-text" from "Goal 1." Once the agent completes its "read-prompt" COPY operation and thereby applies that operator, Soar chunking learns a new apply rule for that instructed COPY. This chunk is shown in the figure as just-learned in procedure memory.

Elaboration contexts are special in the PROP_3 model. They instruct elaboration rules, and elaboration rules in Soar match and fire *in parallel* with each other. This allows the agent to evaluate all condition lines for all retrieved proposal instructions in parallel as a batch as soon as the agent retrieves the procedure context. Without this ability, the agent would need to spend up to a decision cycle per condition line for all proposals per procedure context before it could consider each instructed proposal as a competing decision choice. With this ability, PROP_3 only ever requires a single decision cycle for P3 to evaluate all operator proposal instructions. This also means that there is no more practical benefit from compiling condition PRIM rules together. In PROP_3, P5 is only needed to compile action (apply) PRIM rules together.

Figure 9.3 shows an example of the initial "Transcribe Text" elaboration context. Again, the context instructs proposals for three operators, "get-text," "write-text," and "finish." The logic of the conditions for the three operators shown is as follows: Define `slot1` to hold the text that the agent can read from the prompt. (Recall that "COPY-input-to-slot1" places the input in that WM slot.) Propose "get-text" if there are not yet any active slot1 contents in WM, that is, if the

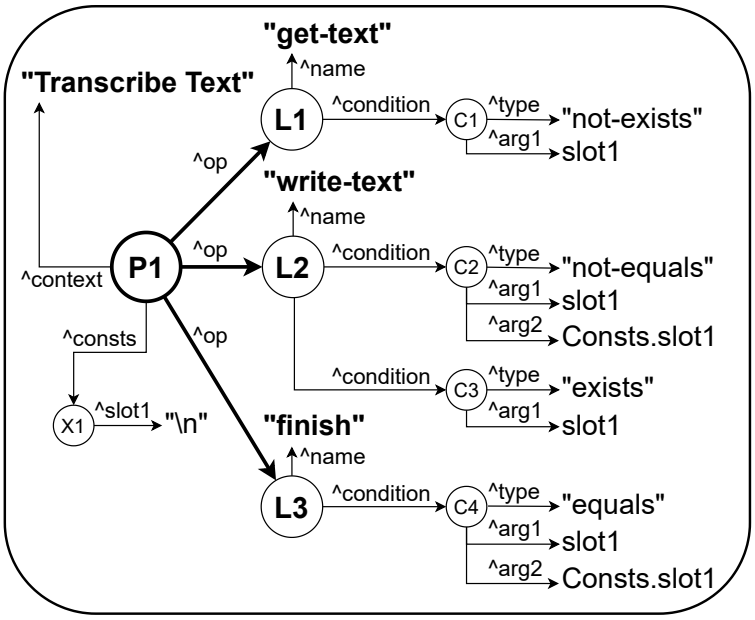


Figure 9.3: The starting "Transcribe Text" elaboration context structure for the transcribe-text task.

agent does not know any input from the prompt in its WM. Propose "write-text" if there is active slot1 content in WM and if that content is not the end-of-line character, "\n." (This assumes that the end-of-line character indicates that there is no more text to write from this prompt.) Finally, propose "finish" if the contents of `slot1` actually are the end-of-line character.

Notice the `consts` link coming from node `P1`, which points to the node labeled `X1`. All elaboration contexts have such a link. The `consts` link keeps the rule-specific values needed for all the condition lines of all the proposals instructed in the elaboration context. In this case, there is only a single slot under `X1` called `slot1`. Condition lines reference this slot as `Consts.slot1` (to distinguish it from the global `slot1`). The `X1` object could point to any number of values, however.

In summary, the use of procedure contexts to model PRIMs theory in Soar presents four key differences with the Actransfer design. First, Actransfer instructions are self-contained bundles of conditions and actions without the hierarchical structure that $PROP_3$ uses. Only one Actransfer instruction can be in WM at a time, whereas $PROP_3$ allows one procedure context hierarchy branch at a time. Actransfer retrieves the instruction with the highest activation.

Second, Actransfer shields decision making from irrelevant procedures by conditioning rule logic on explicit *goal names* in WM, as is standard in ACT-R, Soar, and other architectures. $PROP_3$ shields decision making via the context-specific set of available operator proposals, without the need for explicit goal names. Effectively, the name of the procedure context is the goal name, but this label is more for the sake of the human agent designer than the agent. The $PROP_3$ agent's condition lines do not need to test this name.

Third, Actransfer goal names are like any other WM value, and the Actransfer agent can change them in parallel with other WM operations using rule actions, without additional latency. In the procedure context approach, switching contexts must be a separate serial action with its own performance cost.

Fourth, Actransfer treats condition PRIMs and action PRIMs the same, where a separate decision cycle is needed for each condition PRIM rule in P3 or for each rule that composes PRIM rules. $PROP_3$ represents condition PRIMs as Soar elaboration rules, which means they fire in parallel with each other. This allows the agent to process sets of instructions at once without impacting real-time task performance.

In section 9.4 I show that this approach is sufficient for the behaviors that humans demonstrate in the experimental tasks of the Actransfer suite, with one notable exception.

For a more detailed explanation and discussion of how procedure contexts function in $PROP_3$, see appendix C.

9.1.2 Task Sets

I now describe the theoretical correlate of procedure contexts in human WM research, the task set. The link between PRIMs theory and task sets via procedure contexts is a novel contribution of my work.

The term *task set* (Rangelov et al., 2013; Sakai, 2008) is used in psychology and neuroscience research to describe a mental configuration that corresponds with the intention to do a task. A subject's active task set must change when the subject intends to switch tasks. Task sets are often measured as brain activity patterns in the Pre-Frontal Cortex (PFC) associated with sets of context-specific stimulus-response behavior. Though they are defined abstractly as psychological constructs, they bear startling similarity to what procedure contexts describe computationally (Stearns & Laird, 2020).

The existence of a task set is often measured through neural activity that is sustained while a subject comprehends instructed task behavior, waits to perceive task stimuli, recognizes them, and responds. It is considered to be a WM representation of a set of contextually associated stimulus-response rules, which make behaviors available to decision making (Oberauer, 2010). Studies indicate a hierarchical organization of task sets across the PFC and to some degree the anterior cingulate cortex (Dosenbach et al., 2006), distributed according to the abstractness of the represented procedures (Sakai, 2008). Task sets that deal with more primitive, sensory-based stimulus-response rules are found to activate the posterior PFC, while more abstract, cognitive rules activate the anterior PFC. As people practice rule-based task behavior, their task set activation patterns also have been observed to transition from the anterior to the posterior, indicating a transition from

an early abstract representation into a more concrete form as the subject gains experience (Cole et al., 2013). The hierarchical nature of task sets allows a degree of partial transfer among them (Rangelov et al., 2013).

Neural activity associated with a task set changes when a human switches tasks or switches to different operations within a single task. Specific operations, such as comparing whether two objects are the same, have been linked with specific neurons (Sakai, 2008). The time required to establish a task set is a dominant component of task switch costs and of WM interference. By configuring stimulus-response mappings for current goals, task sets filter the scope of perceived stimuli and available responses, thereby shielding decision making from irrelevant information (Dreisbach & Haider, 2008). The cost of switching to a new task is considered to be a consequence of successful shielding during the prior task.

This description bears striking similarity to the Soar problem-space computational model in three particular ways. First, both task sets and problem spaces describe *sets* of operations (operators) related to a task goal. Second, both are represented *hierarchically*. Third, both serve the function of *shielding* decision making from irrelevant stimulus-response rules (proposals). Procedure contexts appear to be especially similar to task sets in that they are structures in WM that drive the theoretical function of problem spaces in a Soar agent. Procedure contexts also are similar to task sets in that it takes time to switch the active procedure context, according to Equation 9.1.

An additional connection with PROPs research is how task sets are learned. Task sets are theorized to consist of a configuration of the perceptual, attentional, mnemonic, and motor processes necessary to perform a task that is learned after repeatedly practicing and interpreting task instructions received verbally from an instructor (Sakai, 2008). By this description, learning a task set is *cognitive phase* learning (Fitts & Posner, 1967). As I discussed with PROP_2, cognitive phase learning corresponds with the process of learning how to use declarative task instructions to accomplish a task (Stearns & Laird, 2018). I argue that procedure contexts correspond with task sets, and that time required to establish a procedure context in WM corresponds to the time required to establish a task set.

This insight gives me a connection between PRIMs theory and broader ideas surrounding human WM. Specifically, it shows how procedure contexts might be used to model WM interference and the latency of task switching in general. Task set theory describes latency for task switching based on the time required to establish a task set in WM. WM interference occurs when the active contents of WM interfere with (slow down) one's ability to access a different task set. PROP_3's design entails that the time required to establish a procedure context in WM is the decision cycle time required to establish any new subgoals and retrieve procedure contexts from SMEM into the subgoal's WM partition. I can thus model task switching in PROP_3 as the process of switching subgoals, and WM interference as a delay in establishing the desired procedure context in WM.

One direct way to model this kind of interference is through the complexity of the procedure context hierarchy stored in LTDM. Not all procedure contexts are connected directly to each other. If the agent has a chain of procedure contexts linked in the order W->X->Y->Z, and the agent currently only has W in its WM, then it will take twice as long for it to access Y as it will to access X, and three times as long to access Z. The more distant the desired procedure context, the more processing an agent must do before it can invoke the instructed task operations.

In summary, though procedure contexts were designed specifically to address the P2/P3 computation problems, they fundamentally shape the whole of $PROP_3$ processing, contribute new insights for each of the P1-P6 phases, and connect PRIMs theory in Soar with a rich line of neuroscience and decision making research in task sets.

9.1.3 Learned Decision Making

The procedure context approach I described above lets P3 process the conditions of task instructions. P3 must also be able to select an instruction when there are multiple choices. One approach would be to include instructions for preference rules within elaboration contexts. It would then be up to the agent designer to define useful preference rule instructions for each task. However, the agent designs I borrow from the Actransfer experiments do not define preference rules. Another approach would be to use Soar's Reinforcement Learning (RL) to learn P3.

As mentioned in section 6.1, Soar allows an agent to use RL to learn utility-based preferences for operators, similar to the way ACT-R agents learn utility for rules. RL is well established within both artificial intelligence and cognitive science for its ability to replicate human behaviors and even achieve super-human task performance if used in a specialized manner. RL in Soar is an architectural process, and the architecture handles how the agent updates operator utilities based on reward and how those utilities affect decision making, though it is up to the agent designer to choose to enable RL in Soar and to choose specific parameters for the RL processing.

Actransfer does provide environment reward functions for RL in the experiment suite. The Actransfer agents use this to learn rule selection after they have proceduralized their instructions enough to no longer need the PRIMs process. I use this to let the $PROP_3$ agent learn operator selection during the PRIMs process using Soar's RL. In the process, I show how Soar's RL can combine with PRIMs principles to use value function features that are both task-independent and transferable. In section 9.4, I show that this approach achieves human-like learning patterns in some tasks.

Soar represents utility values for operators by including them in the agent's *numeric-indifferent* preference rules. An example numeric-indifferent preference rule could be, "IF ('get-text' is proposed AND the prompt is flashing) THEN (add utility 0.6 to 'get-text')." The condition-action

structure of these rules must be provided by the agent designer, but the architecture adjusts the numeric utility value over time based on environment feedback. For example, if the agent learns from experience that it generally gets little (or negative) reward from the action "get-text" when the prompt is flashing, it can modify the example rule above into, "IF ('get-text' is proposed AND the prompt is flashing) THEN (add utility -0.03 to 'get-text')." A numeric-indifferent preference rule that has its utility value updated over time is called an RL rule in Soar.

When agent designers use Soar RL rules, they define the value function features by how they design the condition/action structure of those rules. In the example rule above, "the prompt is flashing" is the environment feature that the programmer builds into the agent, and which Soar will learn utility for over time. But this feature is only relevant in a task that can have a flashing prompt. To make $PROP_3$ task-independent and able to use Soar's RL to apply P3 decision making for PRIMs theory, I need to make the value function features task-independent.

I observed that the union of PRIMs theory and Soar provides two classes of features that RL rules can condition on and that can make RL in $PROP_3$ truly task-independent and task-relevant. These two classes of features correspond with the *conditions* and *actions* of any decision. First are the general *condition line primitives* used for an operator proposal. These can transfer across decision making the same way that PRIMs can. Second are the names for the specific *procedure contexts* that each proposal points to, which the agent would retrieve if it selected each proposed operator. While individual procedure contexts are usually specific to a particular task, the fact that procedure context names are invoked in all tasks means that the agent can use the same primitive RL rule base for all procedure context names.

$PROP_3$ uses primitive RL rule templates[1] to generate RL rules for each condition line and proposed operator name on-line as they are experienced. Once generated, the agent can thereafter learn utility values for each based on reward from the environment. Initially, each RL gives a utility of 0.0. For example, consider the elaboration context shown in Figure 9.3 on page 114, and the conditions for the "write-text" operator proposal. There are two task-independent condition lines, "(`slot1 <> Consts.slot1`)" and "(`slot1` exists)," plus the single name of the operator that would be proposed if these conditions were satisfied, "write-text." Thus, the first time the agent retrieves this elaboration context into WM it will create the three RL rules, "IF (`slot1 <> Consts.slot1`) THEN (add 0.0 utility to the proposed operator)," "IF (`slot1` exists) THEN (add 0.0 utility to the proposed operator)," and "IF ("write-text" is proposed) THEN (add 0.0 utility to the proposed operator)." After creating these three rules, the architecture can adjust the utility values for each based on ongoing experienced reward.

Similar to ACT-R's mechanic that increases the utility of learned rules that are frequently prac-

[1]Template rules are a feature of Soar for generating rules based on a primitive template rule. For further details, see the Soar literature.

ticed, I built PROP$_3$ to generate intrinsic reward for completing an instruction. The more often the agent practices a particular instructed operator, the greater its utility for that operator.

The utility associated with a condition line represents the utility associated with considering the particular environment feature during decision making. The utility associated with a named procedure context represents associated with a proposed action. Thus these two classes of RL rules are general across all condition/action behavior that could be represented in Soar. What if an agent designer wished to associate a particular utility with a particular conjunction of conditions, such as "IF (the prompt is flashing and the text has been transcribed) THEN (add 1.0 utility to the proposed operator)" or "IF (the prompt is flashing and the text has not been transcribed) THEN (add -1.0 to the proposed operator)?" This could easily be done in PROP$_3$ by having an instructed elaboration create a new WM element for the conjunction of conditions, and then associating the desired utility with that new WM element. This would turn the above rules into "IF (flashing-and-transcribed) THEN (add 1.0 to the proposed operator)" and "IF (flashing-and-not-transcribed) THEN (add -1.0 to the proposed operator)," where "flashing-and-transcribed" and "flashing-and-not-transcribed" refer to a single WM element with that name.

It should be noted, however, that the use of condition lines as value function features does not inherently support the ability for the agent to be influenced by features that are not part of a proposed operator's conditions. For an environment feature to influence PROP$_3$ decision making, that feature must be included in the condition lines of the relevant operators in some form. I discuss this question further in appendix section C.3.2.

The approach I described for RL in PROP$_3$ is task-independent, and it solves the problem of defining how P3 can be learned in Soar. To my knowledge, Soar RL has not been used with task-independent, transferable RL rules in the manner of PROP$_3$ before.

There are many ways that one could explore the merits and trade-offs of this type of RL learning with respect to the field of RL research as a whole. For example, how well does this approach compare with state-of-the-art RL algorithms for basic or challenging benchmark tasks in AI, and what kind of constraints might this approach impose on an AI agent design? However, my thesis focuses on modeling human procedural learning with Soar and PRIMs theory, so these questions are outside the scope of this present work.

9.2 Problem 2: Task-Independent Timing

The problem of P6 is how a modeler can ensure a robust, reliable, task-independent function for mapping computation to human time. If the function used is not reliable, then it is difficult to draw conclusions about the quality of the model.

I discussed in section 4.3 how Actransfer uses $T_{retrieve}$ (Equation 4.1) from ACT-R in such a

way that a task-specific parameter F_r changes the way that computation maps to time for each task. F_r is not consistent across models for different tasks, and while this might be common in the ACT-R tradition, I believe this prevents meaningful conclusions about any model that is supposed to be *task-independent*. Further, it means that much of the evaluation of Actransfer's procedural learning model is based on a declarative learning function. Thus, I marked P6 in red in Figure 9.1.

In PROP$_3$ I remove $T_{retrieve}$ from the model's task behavior. $T_{retrieve}$ is not task-independent and becomes a confounding variable when evaluating the effects of PRIMs processing. It is true that PRIMs processing should affect the timing of the model's declarative processes. Declarative retrievals should take some amount of time, and compiling PRIM rules results in fewer declarative retrievals, and this should therefore reduce model timing. However, the problem with $T_{retrieve}$ for my work is that it is not a constrained enough model of timing to allow me to isolate the effects of PRIMs processing so long as it is present.

I therefore derive timing for PRIMs processing in PROP$_3$ only from the number of processing cycles required by the model. I believe this clarifies the boundaries between what PROP$_3$'s procedural learning computation can and cannot model while also satisfying D3 for task-independence.

I time PROP$_3$ using 50 msec per decision cycle. Soar requires a decision cycle to propose and select an operator, a decision cycle to enter a subgoal, and two decision cycles to query for and receive a procedure context structure from SMEM. These three behaviors supply all timing for the PROP$_3$ agent's cognitive processing. Besides this, I only add the environment interaction time such as for motor actions, using the same times as used for the Actransfer experiments.

While removing $T_{retrieve}$ significantly reduces the time observed in the model's task performance, procedure contexts introduce additional processing that will add to it. I call the effects of this processing *problem-space latency*. I define problem-space latency as the total time required for a Soar agent to transition between problems spaces.

Entering a problem space requires the agent to create a subgoal and retrieve a procedure context into WM. These both require decision cycles. A PROP$_3$ agent will need more decision cycles for tasks with deeper problem space hierarchies because it will have to enter more subgoals and retrieve more procedure contexts. Similarly, the agent will require fewer decision cycles for tasks with shallow problem space hierarchies, where most of the instructions are contained within only one or two layers of procedure contexts. PROP$_3$ contrasts with Actransfer in that the simulated time depends as much on the hierarchical task structure as on the number of primitive operations.

I formally define problem-space latency with the following function:

$$T_{ps} = F_g \times G + F_c \times C \tag{9.1}$$

G is the number of times an agent enters a subgoal, F_g is a parameter for the time required to enter a subgoal (50 msec by default), C is the number of times the agent retrieves a procedure

context, and F_c is a parameter for the total time required for each procedure context retrieval (100 msec in Soar). G and C increase or decrease based on how many branches the agent traverses from its procedure context hierarchy. F_g and F_c are constant across tasks. The entire function simply describes a fixed amount of time per subgoal and per procedure context retrieval.

In the earlier "Transcribe Text" example of Figure 9.2, the agent started from the "Transcribe Text" goal and retrieved the "Transcribe Text" procedure context for that problem space. Then it proposed and selected the "get-text" operator, proceeded to the "get-text" subgoal, and retrieved the "get-text" procedure context. Then it proposed and selected the "read-prompt" operator, proceeded to the "read-prompt" subgoal, and retrieved the "read-prompt" procedure context. Finally, it applied the "read-prompt" task operation with the "COPY-input-to-slot1" operator. In this sequence, the agent selected 3 operators, "get-text," "read-prompt," and "COPY-input-to-slot1," for 3 decision cycles. The agent entered 2 new subgoals, "get-text" and "read-prompt," for 2 more decision cycles. And the agent needed to retrieve three procedure contexts for "Transcribe Text," "get-text," and "read-prompt," for 6 decision cycles. Thus, this whole sequence requires 11 decision cycles in Soar. Using 50 msec per cycle, this sequence of cognitive operations takes the agent 0.55 sec. Of the 11 cycles, 8 are from entering a subgoal or retrieving a procedure context. Thus, 0.4 sec of the 0.55 sec is from problem-space latency.

Chunking affects PROP_3 timing by reducing and eventually eliminating the need for the agent to use apply contexts for action lines. After chunking the "COPY-input-to-slot1" application of "read-prompt" as in Figure 9.2, the agent can apply the "read-prompt" operator straight from "Goal 2." This reduces the agent's total sequence to 7 decision cycles, or 0.35 sec. But chunking in PROP_3 does *not* replace the need for the agent to use elaboration contexts across subgoals, such as the "Transcribe Text" or "get-text" contexts, since these give structure to the sequence of agent operations and cannot be summarized together. (Even an expert typist never loses the need to visually find text before transcribing it.)

In Equation 9.1, G and C are equal for any task in PROP_3. I keep these as separate terms, however, because technically by Soar theory it would be possible to retrieve multiple procedure context structures for a single subgoal over time, and this would make C greater than G. This is not how PROP_3 functions for my model of PRIMs theory, but might be relevant for future research that builds on the PROPs system.

While the PROP_3 process of entering a subgoal might be compared to the Actransfer process of changing the goal name kept in its WM **goal** slot, Actransfer does not allocate separate processing or timing for goal changes. As with the modern ACT-R approach, an Actransfer agent modifies its goal by changing the WM value the way it would change any other WM value. Goal changes are instructed with action lines and therefore compiled to be executed in parallel with other actions. For example, an Actransfer agent might have an instruction with two action lines, "(`input ->`

`output` AND `const1 -> goal`)," and these could both be compiled together. In PROP$_3$, these two operations are always done serially.

If treating procedure contexts as a model of task sets, T_{ps} serves as a simple, task-independent model of task set switching costs. The way in which T_{ps} must be serial with instructed rule actions aligns with task set theory. A new task set cannot replace another task set in WM until the other task set is no longer being used to drive behavior.

9.3 Three-Phase Parameters

As discussed in section 8.5, evaluation with PROP$_2$ revealed the importance of considering the cognitive phase during experimental design. A theoretical conclusion from that work was that, as a cognitive phase process, the process of learning P2 must be inherently declarative in nature. For the agent to learn how to retrieve declarative task instructions, it must in some form learn declarative associations. This means I ought not model it primarily with procedural learning. Because PRIMs theory and the PROPs system are meant to be models of procedural learning, I therefore do not attempt to include cognitive phase (`Learned`) learning in the PROP$_3$ design.

It is worth noting that the way PROP$_3$ intimately connects P2 with P3 and the hierarchical PSCM structure of a task would make cognitive phase learning in PROP$_3$ even more purely declarative in nature than it was in PROP$_2$. For the PROP$_3$ agent to learn P2, it would need to learn the procedure contexts structures themselves. It would need to learn the hierarchical connections between them as well as even the declarative condition lines that allow an agent to progress from one to another. It is far beyond the scope of my thesis to model how an agent might create or modify its declarative understanding of task structure in this manner.

Though PROP$_3$ does not implement cognitive phase learning, this does not negate the contributions of the PROP$_2$ model. PROP$_2$ modeled the cognitive phase in the context of Actransfer's approach for P2 and Taatgen's hypothesis for a more comprehensive implementation of P2. Rather, PROP$_3$ shows how a model of PRIMs that is fully consistent with Soar in its implementation does not easily support that kind of learning at the present time. PROP$_2$ was better able to model the cognitive phase due to the ways it still differed from Soar theory in its design for P2 and P3.

Experimentation with PROP$_1$ and PROP$_2$ showed that `Auto` learning is not usually a good parameter for modeling the kinds of tasks explored in this thesis. As previously discussed, later experimental revisions of Actransfer disabled this learning behavior for computational reasons. The theoretical connection with three-phase theory introduced during work with PROP$_2$ also shows that `Auto` learning would make sense more with primarily motor tasks that humans can learn to do without cognitive effort, and not with the sort of cognitive tasks that are studied here, such as arithmetic or text editing. Some measure of cognitive effort always ought to be necessary for these

tasks in humans.

`Auto` learning is also not easily supported by $PROP_3$ due to the nature of procedure contexts. To chunk `Auto` rules, the agent would have to chunk the processing of elaboration contexts, such that the agent would then no longer need the elaboration context instructions. A PROPs agent could not be based upon the Soar PSCM without something along the lines of elaboration contexts. It would take a remarkably different programming and architectural paradigm to support `Auto` learning in Soar at this time. The exception might be the case where task instructions truly required only a single layer of elaboration context. In that instance, a hierarchy of elaboration contexts is redundant, and a $PROP_3$ agent could learn to use `Auto` chunks without the need for any procedure contexts. As it stands, and for the reasons given above, $PROP_3$ does not attempt to support `Auto` learning.

9.4 Evaluation

$PROP_3$'s solutions for both P2/P3 and P6 come from the addition of procedure contexts. Because procedure contexts represent a single computational change between $PROP_2$ and $PROP_3$, I must evaluate these solutions together.

I test $PROP_3$ by applying it in a task-independent manner across the whole Actransfer experiment suite, adding the WM/Stroop task and the task-switching task alongside the arithmetic and editors tasks. I divide the suite of four experiments into two main evaluation blocks, as published by Stearns and Laird (2020), due to the different characteristics of the tasks.

The first evaluation replicates the arithmetic and editors tasks as I did when evaluating $PROP_2$. These tasks test procedural learning curves as subjects get faster at their tasks with practice. They also each represent different time scales for task processing, where human subjects in the arithmetic task took 2 to 12 seconds to complete a set of calculations while humans in the editors task took 20 to 120+ seconds for each assigned operation. I want to evaluate whether $PROP_3$'s task-independent implementation can model both time scales as a single system.

The second evaluation replicates the remaining two experiments from the Actransfer suite, the WM/Stroop experiment and the task-switching experiment. These experiments do not directly test procedural learning curves or transfer like the first two, but rather test transfer in decision making, task-switching latency, and WM interference. These two experiments allow me to test the robustness of using $PROP_3$ procedure contexts as a model of task sets in the manner I described at the end of section 9.1.2.

In this chapter I use "evaluation" to refer to one of these two blocks, "experiment" to refer to one of the four human studies replicated in the Actransfer suite, and "task" to refer to one of the specific tasks within one of these experiments. The WM/Stoop and task-switching experiments

both involve multiple tasks.

Because PROP$_3$ is a model that does not incorporate any kind of $T_{retrieve}$, I compare my model behavior with Actransfer when Actransfer $T_{retrieve}$ time is omitted in addition to the original Actransfer model. Anywhere that PROP$_3$ comes short in replicating the trends of human performance implies that the trends come from declarative learning or some other process besides the procedural learning of PRIMs processing.

It is difficult to make predictions about PROP$_3$ performance. The way PROP$_3$ evaluates condition lines in parallel should make it faster compared to PROP$_2$ or Actransfer, because P3 will only ever require a single decision cycle. The exact gains from this will depend on the instructions for each task. If the instructions from Actransfer use many conditions, the gains for processing conditions in parallel in PROP$_3$ will be greater. The nature of T_{ps} could make PROP$_3$ take longer, however, if the hierarchical goals in the instructions have a significant depth. The greater the depth of the procedure context hierarchy, the more time required to retrieve the deeper procedure contexts.

I once again use the same agent instructions used for Actransfer. However, since Actransfer instructions are not explicitly hierarchical, I modify them slightly to convert them to a procedure context hierarchy. I group all Actransfer instructions that test the same goal name into a single procedure context, and replace each goal name change with a procedure context switch (either into a new subgoal or returning to a parent goal). Otherwise, the instructions are unchanged except where noted.

9.4.1 Evaluation 1: Hierarchical Procedure Composition

For the arithmetic and editors tasks, I use the same environment setup and agent parameters as for PROP$_2$. I again used $\theta_p = 10$ for the arithmetic task, and $\theta_p = 48$ for the editors task.

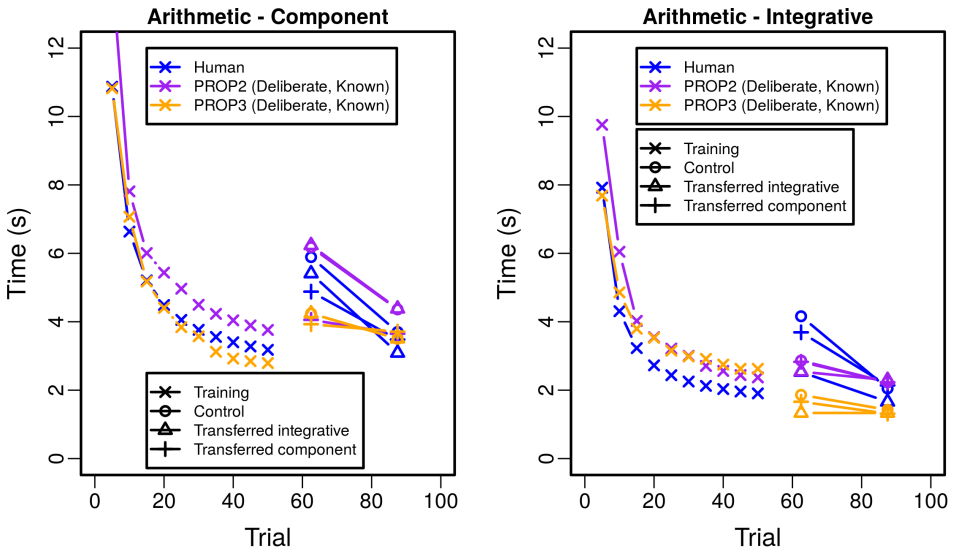
The arithmetic problem space hierarchy is a fairly simple one. The starting problem space instructs proposals for each main step in the memorized sequence of mathematical operations, such as "subtract-a-from-c" or "multiply-b-and-3." Each mathematical operation is then carried out only one subgoal deep in the hierarchy.[2] The shallow hierarchy implies a low T_{ps}.

The editors task instructions written for Actransfer are more complex. They use many goals and subgoals to define task reasoning, such as "find-edit-line," "move-cursor," "type-word," and so on, for a problem space depth of up to six subgoals when translated into PROP$_3$ procedure contexts. This implies a higher T_{ps}.

[2]Arithmetic operations such as *subtract-a-from-c* could conceivably be broken further into subgoal actions if the math operations were computed manually, but this agent design assumes that adults have elementary math problem answers memorized and do not need to compute them manually.

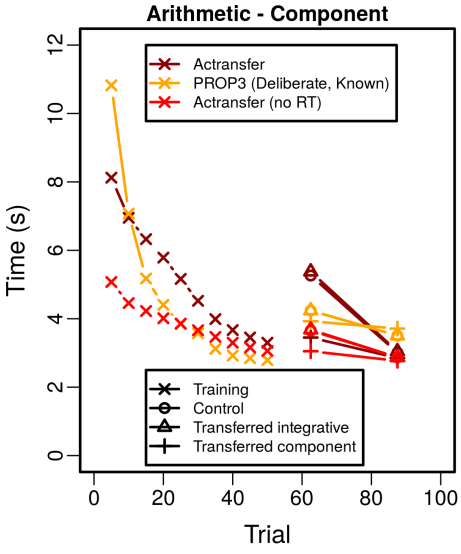
9.4.1.1 Arithmetic Experiment

Figure 9.4 shows arithmetic task performance by humans and by the Actransfer, $PROP_2$, and $PROP_3$ models. PROPs models are on the top, Actransfer on the bottom. Actransfer results that omit time from $T_{retrieve}$ are labeled `(no RT)`. Component step results are on the left and integrative step results on the right.

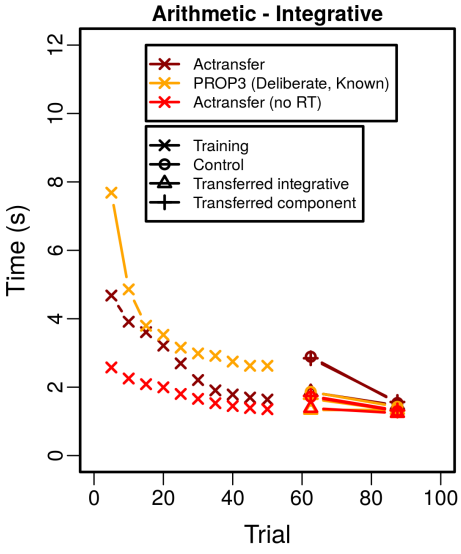


(a) Component steps

(b) Integrative steps



(c) Component steps



(d) Integrative steps

Figure 9.4: Human, Actransfer, $PROP_2$, and $PROP_3$ performance for the arithmetic task.

First, observe that $PROP_3$ shows slightly faster overall timing than $PROP_2$ in the top two fig-

	Component Steps			Integrative Steps		
Model	r^2	*MAE*	*MAPE*	r^2	*MAE*	*MAPE*
Actransfer	0.808	0.84	15.7	0.758	**0.58**	**14.2**
Actransfer (No Retrievals)	0.844	1.01	13.6	0.777	1.28	34.5
$PROP_1$ (Deliberate, Known)	0.994	2.71	61.3	0.969	2.05	72.2
$PROP_2$ (Deliberate, Known)	**0.995**	1.04	20.2	0.991	0.88	28.6
$PROP_3$ (Deliberate, Known)	0.992	**0.27**	**7.12**	**0.997**	0.65	26.9

Table 9.1: $PROP_3$ goodness-of-fit measures for the arithmetic task. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

ures, which brings the model closer to human performance, particularly for component steps. As described earlier, this speedup comes from $PROP_3$'s ability to evaluate condition lines in parallel without using more than one decision cycle at a time for P3.

Second, notice that $PROP_3$ exhibits more transfer than humans, as seen in the faster performance at the start of the various transfer conditions for trials 51-100. To better understand this, compare $PROP_3$ performance with that of Actransfer in the lower two figures when Actransfer also omits $T_{retrieve}$. Notice that the shape of the transfer data for $PROP_3$ is fairly identical. A significant portion of Actransfer's early latency in the transfer trials (51-100) was from $T_{retrieve}$ due to the lower initial declarative activation in the instructions used for those trials. The fast performance when omitting $T_{retrieve}$ at the start of the transfer conditions indicates a very high amount of procedural transfer. Inspection of the processing shows that this is due to how the Actransfer model design treats identical calculations as identical subgoals, which can be transferred fairly completely. For instance, the goal *subtract-a-from-c* is carried out with the same instructions for any use of that operation in any mathematical routines. Slower human performance in the transfer conditions (less transfer) might indicate that humans require significant declarative learning time when switching to the transfer conditions, in the manner modeled by Actransfer, or it might indicate that humans do not mentally represent operations in quite so modular a fashion as the Actransfer and $PROP_3$ instructions do. This latter interpretation is consistent with Elio's (1986) findings. In that study, changes to the integrative structure and order of the algorithm reduced component step transfer, implying that the subjects' skill representations for the individual math operations were not modular but depended on the structure of the entire algorithm.

Table 9.1 shows the goodness-of-fit measures for the arithmetic training curves for $PROP_3$ and the other models. $PROP_3$ shows significantly lower error across the board than the other models for component steps, and is second only to Actransfer for integrative steps, since it levels off slow compared to humans for those steps. Its component step r^s is 0.003 behind $PROP_2$, but this is a negligible difference. Its r^2 for both component and integrative steps is very high, > 0.99.

Table 9.2 and Table 9.3 show the transfer scores for $PROP_3$ next to the other models. As before,

Component Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		0.48	1.01
Actransfer		-0.11	1.81
Actransfer (No RT)		-0.03	0.60
$PROP_1$ (Deliberate, Known)		-0.04	2.88
$PROP_2$ (Deliberate, Known)		-0.06	2.12
$PROP_3$ (Deliberate, Known)		0.001	0.31
	Percent Transfer		
Human	11.75%	27.38%	44.63%
Actransfer	38.09% (+26.34)	**34.51% (+7.13)**	95.23% (+50.60)
Actransfer (No RT)	53.12% (+41.38)	50.85% (+23.48)	100.81% (+56.18)
$PROP_1$ (Deliberate, Known)	**13.50% (+1.75)**	12.55% (-14.82)	88.72% (+44.08)
$PROP_2$ (Deliberate, Known)	38.36% (+26.62)	36.96% (+9.58)	92.50% (+47.86)
$PROP_3$ (Deliberate, Known)	58.25% (+46.50)	58.28% (+30.90)	**67.26% (+22.62)**

Table 9.2: $PROP_3$ transfer for arithmetic component and integrative steps.

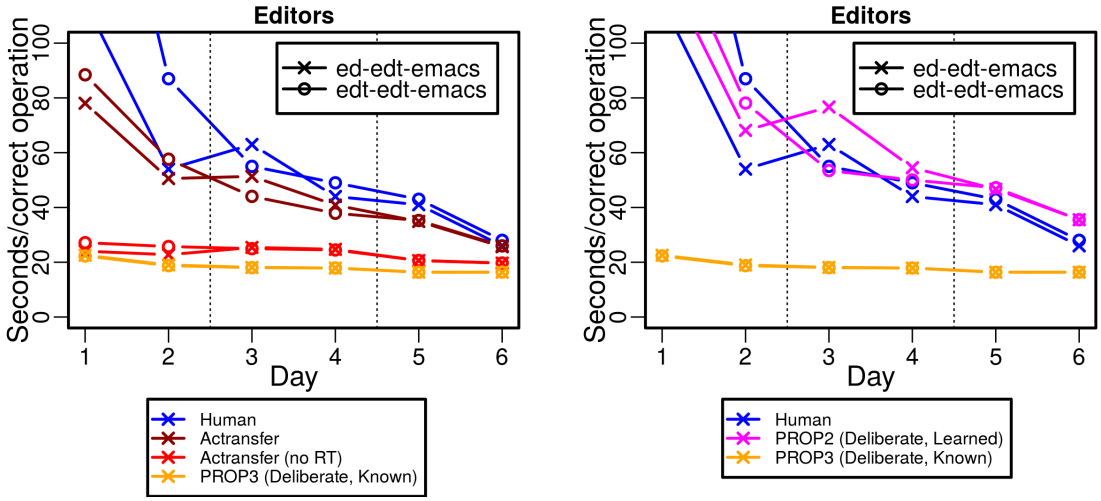
there is no clear pattern for component steps. Actransfer has the closest transfer for integrative steps, however. The fact that $PROP_3$ has the same flatter and faster performance for trials 51-100 as Actransfer `(No RT)` combined with much higher initial latency in the training due to PRIM resolution means that the relative transfer from the start of training to the start of the transfer conditions is much higher for $PROP_3$, especially for integrative steps as shown in Table 9.3.

Overall, though its transfer is not as close as that generated by Actransfer, the $PROP_3$ agent's fit to the human training curve is arguably the best in terms of both shape and error of all the models. Again, this is achieved by using the Actransfer task instructions in procedure context form and by treating Actransfer goal name changes as procedure context retrievals with their own time cost.

Integrative Step Transfer			
Model	*Control*	*Integrative*	*Component*
	Difference from Control		
Human		1.63	0.47
Actransfer		1.03	0.04
Actransfer (No RT)		0.33	-0.02
$PROP_1$ (Deliberate, Known)		0.82	0.06
$PROP_2$ (Deliberate, Known)		0.32	0.04
$PROP_3$ (Deliberate, Known)		0.53	0.20
	Percent Transfer		
Human	-1.62%	71.84%	19.56%
Actransfer	**36.89% (+38.51)**	**89.07% (+17.23)**	**39.14% (+19.57)**
Actransfer (No RT)	53.71% (+55.33)	95.85% (+24.01)	50.65% (+31.08)
$PROP_1$ (Deliberate, Known)	65.34% (+66.96)	89.95% (+18.11)	67.10% (+47.54)
$PROP_2$ (Deliberate, Known)	83.25% (+84.87)	94.21% (+22.37)	84.50% (+64.94)
$PROP_3$ (Deliberate, Known)	138.52% (+140.14)	165.24% (+93.40)	148.68% (+129.12)

Table 9.3: $PROP_3$ transfer for arithmetic integrative steps

9.4.1.2 Editors Experiment



(a) (b)

Figure 9.5: Human, Actransfer, and $PROP_3$ performance for the editors task.

Figure 9.5 shows human, Actransfer, $PROP_2$, and $PROP_3$ results for the editors task. One can immediately notice that the $PROP_3$ model replicates the same scale of performance as

the Actransfer agent that is missing $T_{retrieve}$. It is, in fact, slightly faster even than this Actransfer model. The higher T_{ps} needed for the editors task due to its deeper goal hierarchy adds time that the Actransfer agent does not represent, but this is outweighed by the speed gained from evaluating conditions in parallel.

Due to a fault in the experiment code, Stearns and Laird (2020) reported in error that the PROP$_3$ editors model replicated the higher time scale of the original Actransfer results. They attributed this to the higher T_{ps} and failed to account for the time gained from evaluating conditions in parallel.

The inability of PROP$_3$ to replicate human time scales for this task indicates one of two things. One possibility is that the majority of human processing in this task is not the kind of procedural learning processing shown by humans in the arithmetic task. Actransfer's computation attributed most of it to $T_{retrieve}$, and thus to declarative retrievals. Retrieval times of 0.9 sec for a single PRIMs instruction seems unusual, however. It could be that there are other cognitive processes at work that are not captured by either PRIMs procedural learning or $T_{retrieve}$ declarative learning, however, such as extended deliberation over motor control when using the unfamiliar computer keyboard shortcuts. Another possibility is that the instructions from Actransfer represent the correct types of processing but are too simplistic in the number or structure of operations compared to actual human processing. In PROP$_3$, for instance, one might model higher time scales of behavior using an even deeper procedure context hierarchy. That kind of model would assert that humans require time on the order of 60 sec rather than 20 sec because they engage in a substantial amount of processing that does not directly contribute to performing the task. For example, the slower performance of humans learners might be because they double-check their keyboard actions against the prompt directions multiple times before committing to task actions. In PROP$_3$, this would be modeled with additional procedure contexts for the cognitive operations of double-checking as well as additional time for the motor/vision actions of looking back and forth between screen and directions. However, one would expect human learners to gain confidence with practice and not need to double-check their answers as often over time. This would require the PROP$_3$ agent to restructure its procedure contexts on-line over time, a type of declarative learning.

Figure 9.6 shows what PROP$_3$ performance looks like in this task when the $T_{retrieve}$ from Actransfer is added to its performance, labeled `(+RT)`. Overall timing and transfer trends are similar to the Actransfer model. As with the `(no RT)` models, PROP$_3$ is slightly slower. This again follows from the ability of PROP$_3$ to process conditions in parallel.

Table 9.4 shows the goodness-of-fit measures for Actransfer and PROPs models of the editors task. PROP$_2$ remains the best model of humans quantitatively by these measures. PROP$_3$ measures with or without added retrieval time are overall comparable to those of Actransfer under the same conditions. PROP$_3$ has slightly worse error due to being slightly faster than Actransfer.

Table 9.5 shows the transfer measures for PROP$_3$ and the other models. A model's transfer is

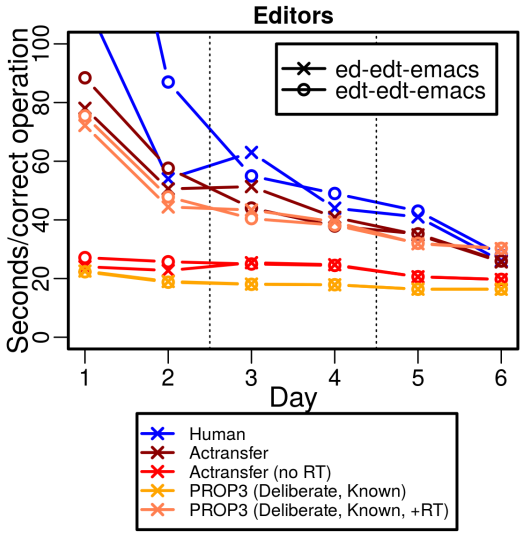


Figure 9.6: Human and model performance for the editors task when $T_{retrieve}$ is added to PROP$_3$.

	ED-EDT-EMACS			**EDT-EDT-EMACS**		
Model	r^2	*MAE*	*MAPE*	r^2	*MAE*	*MAPE*
Actransfer	0.965	10.27	13.4	0.958	31.13	26.7
Actransfer (No RT)	0.299	34.30	52.4	0.534	55.56	57.3
PROP$_2$ (Deliberate, Known)	**0.977**	**5.86**	**10.8**	**0.991**	**24.83**	**21.6**
PROP$_3$ (Deliberate, Known)	0.919	38.79	62.2	0.942	61.04	67.0
PROP$_3$ (Deliberate, Known, +RT)	0.967	15.01	22.5	0.982	36.06	32.0

Table 9.4: PROP$_3$ goodness-of-fit measures for the editors task. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

calculated relative to its own performance in various task conditions, so PROP$_3$'s transfer performance as shown is still a useful metric when examining its procedural learning. Actransfer, PROP$_2$, and PROP$_3$ each are closest to human transfer for three of the transfer conditions. PROP$_3$ comes in second behind PROP$_2$ for many transfer scores, however, superior to the transfer of Actransfer `(no RT)` in most cases except for when transferring to EMACS. This indicates that while the scale of the PROP$_3$ agent's performance is off, and other timing factors must be at work in humans, its procedural learning properties are still comparable to those of the original Actransfer model.

These experiments demonstrate both the strengths and limits of PRIM theory as modeled in both Actransfer and Soar. The procedure context approach for implementing PRIMs theory in Soar is especially able to capture power-law learning at the same time scale as humans in the arithmetic task. Current models are not, however, able to capture human performance in the editors task as a result of PRIMs procedural learning while also satisfying my desiderata. Actransfer replicated editors performance by adding a substantial task-specific $T_{retrieve}$, which violates D3 for task-independence. PROP$_2$ used a hybrid of ACT-R and Soar architectural principles which were

Model	*Global*	*Planning Move to Line*	*Planning Edit Text*
ED to EDT Editors Transfer			
Human	94.97%	87.50%	104.62%
Actransfer	83.54% (-11.42)	82.03% (-5.47)	88.35% (-16.27)
Actransfer (No RT)	79.33% (-15.64)	81.02% (-6.48)	79.58% (-25.04)
$PROP_2$ (Deliberate, Known)	86.95% (-8.02)	**85.88% (-1.62)**	88.47% (-16.15)
$PROP_3$ (Deliberate, Known)	**98.04% (+3.07)**	100.04% (+12.55)	97.63% (-6.99)
$PROP_3$ (Deliberate, Known, +RT)	91.61% (-3.36)	95.14% (+7.64)	**98.11% (-6.51)**
EDT to ED Editors Transfer			
Human	97.18%	91.84%	99.09%
Actransfer	90.85% (-6.33)	89.93% (-1.91)	91.80% (-7.29)
Actransfer (No RT)	85.47% (-11.72)	87.10% (-4.74)	83.79% (-15.30)
$PROP_2$ (Deliberate, Known)	**96.22% (-0.96)**	**92.65% (+0.81)**	102.23% (+3.14)
$PROP_3$ (Deliberate, Known)	100.62% (+3.43)	99.97% (+8.13)	**99.39% (+0.30)**
$PROP_3$ (Deliberate, Known, +RT)	94.47% (-2.71)	93.78% (+1.94)	99.94% (+0.85)
EDT/ED to EMACS Editors Transfer			
Human	64.81%	61.03%	62.29%
Actransfer	**63.41% (-1.40)**	**66.70% (+5.66)**	**57.26% (-5.03)**
Actransfer (No RT)	42.24% (-22.57)	42.66% (-18.37)	36.28% (-26.01)
$PROP_2$ (Deliberate, Known)	84.07% (+19.25)	91.04% (+30.01)	126.92% (+64.63)
$PROP_3$ (Deliberate, Known)	99.55% (+34.74)	100.07% (+39.04)	98.62% (+36.33)
$PROP_3$ (Deliberate, Known, +RT)	88.72% (+29.91)	100.05% (+39.02)	98.62% (+36.33)

Table 9.5: $PROP_3$ transfer in the editors task, along with differences compared to humans.

inconsistent with either architecture alone, which violates D4 for a consistent architectural model. $PROP_2$ also did not yet address the implementation gaps for P3 and P6, such that it did not satisfy D1 for a comprehensive model.

Thus, while I can generate human performance with $PROP_2$ and even get a better fit compared to Actransfer, consistency with the architectural theory and task-independence suggests this is not quite right. $PROP_3$ is what a consistent, comprehensive model looks like, and it shows there are limitations if we stay honest in the details. But as I demonstrate with evaluation 2, exploring these details also reveals connections with other ideas like task sets that were not part of the original theory and which do provide new insights and modeling capabilities.

9.4.2 Evaluation 2: Rapid Decision Making

The WM/Stroop and task-switching experiments in evaluation 2 test rapid decision making, task switching, and WM interference effects. These are behaviors associated with task sets. I use $PROP_3$ procedure contexts as a model of task sets when replicating these experiments.

As I described in chapter 5, Taatgen (2013) used declarative activation in Actransfer to model decision making transfer and $T_{retrieve}$ to represent WM interference. $PROP_3$ does not use $T_{retrieve}$,

nor does it use activation in P2 for decision making as Actransfer does, and so it is not be able to model either of these original experiments in this same way. However, the link between procedure contexts and task sets does imply a potential model of decision making transfer and WM interference in PROP$_3$.

I model this same WM interference using T_{ps} in PROP$_3$, and model this same decision making learning and transfer through the process of learning to select procedure contexts using RL according to the manner I described in section 9.1.3. In contrast to P2, RL in psychology research has long been viewed as a mechanic for decision making, and it is provided as such in both the ACT-R and Soar architectures. I modify the Actransfer task instructions for these experiments slightly in order to provide a model that satisfies D4 as consistent with decision making and task set theory as well as Soar theory.

Where Actransfer models learned decision making through learned P2, I model this same behavior using learned P3 via RL. Where Actransfer models interference from $T_{retrieve}$ using lower activation, I model the same behavior from T_{ps}. The net effect of these modifications is that in some places I divide Actransfer instructions across multiple subgoals with some hierarchical depth that was not originally present. Otherwise, the PROP$_3$ task instructions are equivalent to those of Actransfer. I describe the differences in goal hierarchy for each experiment in the coming sections.

While I consider using P2 to model human decision making to be inconsistent with prevailing theory, I make no objection to using $T_{retrieve}$ to model WM interference. However, the fact that $T_{retrieve}$ adds time that is not otherwise reflected by agent processing cycles means that this approach has relatively little explanatory power, though it does attribute interference effects to declarative activation and learning. An internally-consistent PROP$_3$ model that satisfies D4 implies a different model for interference.

For these experiments, I set Soar's RL parameters to use softmax selection, a learning rate of 0.02, and a discount factor of 0.775, based on a brief sweep of the Soar parameters for the WM/Stroop experiment. I use the same parameters for all tasks in these experiments.[3] I repeated the parameter sweep for the task-switching experiment and found that the same parameters were desirable in those tasks as well.

The tasks in these experiments are different from those of the first evaluation in that they are timed rapid-response tasks, where a cue is shown on a prompt and the agent must immediately respond with a single answer. For example, one task in the second experiment is a flashcard-style task in which the agent must respond whether there are one or two items shown on the prompt. In the Actransfer models for these tasks, the agents must initially take a long time to respond to each such prompt until they learn rules that can perform task responses. In the Stroop task

[3]The Actransfer agents in the test suite also rely on specific RL and LTDM parameters for each task. These differ for each task in each of the original experiments.

of the second task-switching experiment, humans took around 1 sec to respond to task prompts, while the Actransfer agent initially required around 2 sec and eventually learned to respond to prompts in 1 sec. (This difference does not manifest in the evaluations because these experiments measure differences, such as the difference between responding to congruent or incongruent Stroop prompts.) Human subjects on average do not significantly speed up in the Stroop task more than about 0.1 sec within a session of practice, however (Martin et al., 2016), and most studies such as those of Chein and Morrison (2010) and Karbach and Kray (2009) for this evaluation do not bother to report improvements in Stroop performance over time for single task conditions. The relatively steady rapid-response behavior of human subjects implies that humans already possessed the skills needed for rapid responses. I therefore set θ_p to 1 for the tasks in this evaluation. This means that the PROP_3 agent will fully learn its task rules within its first few trial prompts. Like humans, it can start its Stroop task performance around 1 sec and will not speed up much more than about 0.1 sec when practicing a single task condition. Effectively all model learning in these experiments comes from the agent modifying its decision making routine rather than from performing the same routine more quickly.

9.4.2.1 WM and Stroop Experiment

Recall that in the original human experiment, subjects demonstrated reduced WM interference in the Stroop task after a long period of training in a WM span task. Stroop interference in this experiment was measured as the difference in response time between congruent and incongruent stimuli in the Stroop task. After WM training, subjects were able to respond to incongruent stimuli more quickly, closer to their speed for congruent stimuli.

As described in section 5.3, the Actransfer model for this experiment uses $T_{retrieve}$ to explain interference. If the agent is shown incongruent stimuli, it still always retrieves the correct answer from LTDM, but the activation for the correct answer is not as high as when stimuli are congruent. $T_{retrieve}$ models longer retrieval time when the retrieved memory has lower activation, and thus the agent is faster when given congruent stimuli.

The Actransfer agent improves its performance by choosing a "prepare" operation in between task prompts. The "prepare" actions send a command to the architecture's output system so that the agent does not receive the text word as input when the next prompt appears. In theory, this is the agent focusing on just the color concept so that it does perceive the text details. The Actransfer agent can also instead choose to wait idly until the next prompt appeared instead of preparing, in which case its input gives both color and text stimuli to WM.

The WM span training task includes a mandatory "prepare" operation. Recall that in this training the agent is supposed to remember a sequence of items shown in brief prompts. In between prompts, the agent always chooses to prepare by rehearsing the sequence of currently-known items.

The instruction name for the prepare operation was different, but the primitive condition and action lines were mostly the same as for the optional "prepare" operation in the Stroop task. Thus, when the agent practices "prepare" in the WM training task it increases the activation of those condition and action lines, and by spreading, it thereby also increases the activation of the "prepare" instruction in the Stroop task. Thus, the agent is more likely to prepare in the Stroop task after training, and thus demonstrate reduced interference.

PROP$_3$ instead models interference via T_{ps}. The agent has two choices of answer for each color/word prompt. It can answer the text word, or it can answer the font color. Each of these responses is instructed in a different procedure context. The procedure context for answering the text word is only one subgoal away from the starting goal, whereas the context for answering the font color is two subgoals away, and thus not as readily accessed. It therefore takes longer for the agent to access the procedures that would let it answer with the font color than to answer with the text word. This aligns with a model of task sets in which the subject must go through more processing steps to switch to the task set it needs for a particular response.

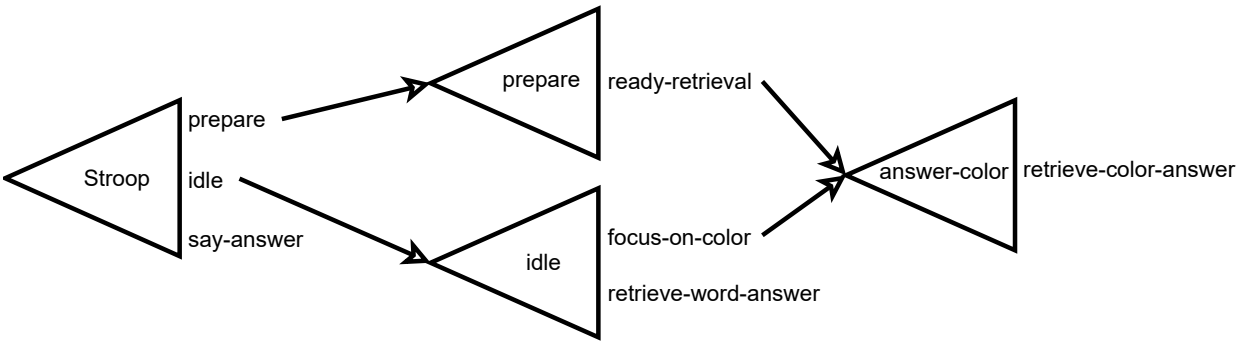


Figure 9.7: The hierarchical goal design of the PROP$_3$ model for the Stroop task.

Figure 9.7 depicts the subgoal/procedure context structure I used, in problem space notation. We replaced the "prepare" action of the Actransfer agent with a "prepare" procedure context that is also available to the agent from the starting context. In between task prompts, the agent can choose either the `prepare` or the `idle` operators, which each lead to their own subgoal and procedure context retrieval. From the "prepare" subgoal, the agent prepares by immediately selecting and entering the "answer-color" subgoal and retrieving that procedure context. The "answer-color" context instructs the agent to answer the prompt font color once text appears. Retrieving the procedure context ahead of time saves time once the agent actually sees the prompt, for a faster response. Alternatively, the agent can select the `idle` context. From that context, the agent can immediately select the `retrieve-word-answer` operator as soon as a prompt appears in order to answer the text word, assuming the prompt text and color are congruent. But if the prompt text and color are incongruent, the agent must instead select `focus-on-color` to retrieve the

procedure context that will let it answer the prompt color. If the prompt text and color are congruent, `idle` is optimal for the agent, because it lets the agent answer immediately without the additional cognitive effort of entering the deeper "answer-color" subgoal. But if the prompt text and color are incongruent, `idle` makes the agent take longer, because it does not benefit from proactively retrieving the "answer-color" procedure context. As soon as the agent has used either the `retrieve-word-answer` or the `retrieve-color-answer` operators to retrieve an answer, the `say-answer` operator in the main task context lets the agent report that answer.

Conceptually, the PROP$_3$ agent learns to prepare in the same way as the Actransfer agent, but the effect of training is not to increase the activation of the "prepare" procedure context but to increase the *utility* assigned to the "prepare" operator. As I described in section 9.1.3, the PROP$_3$ agent uses conditions and actions of an elaboration context as value function features. The agent gets a small amount of reinforcement for operators that it practices frequently. As the PROP$_3$ agent practices the WM training task, it learns to assign a higher utility to operators that rely on the condition lines of the practiced "prepare" instruction and to operators that propose the same "prepare" action. This then transfers to the Stroop task. As in the original Actransfer model, it also get a larger amount of reward for correct task answers. The temporal-difference learning of RL means that the agent will also increase the utility for practiced operators more if they lead it to a correct answer more quickly.

Note that in this design, for both Actransfer and PROP$_3$, WM training is more effective at teaching the "prepare" choice than practicing the Stroop task alone would be. This is because the agent is able to still get correct answers on the Stroop task when it does not prepare, and when this happens this increases the probability of repeating the choice to not prepare in the future, even if this increase is not as great as that for the "prepare" choice when "prepare" is practiced.

The Actransfer model added 0.2 sec for each action of reporting an answer as well as for each cognitive action of focusing on the correct answer. The PROP$_3$ model also adds 0.2 sec for reporting each answer but does not add extra time for focusing on the correct answer. In PROP$_3$ this time comes through T_{ps}.

Figure 9.8 shows the PROP$_3$ model results for the experiment alongside human and Actransfer model performance. PROP$_3$ is similar to the original Actransfer model, despite not using $T_{retrieve}$. Both models are able to improve more in interference when given training than when not given training. Both models show about the same relative improvement when without training, when taking error bars into consideration. However, the PROP$_3$ agent starts with less interference than Actransfer before training, and its interference after training is closer to human performance.

Table 9.6 shows goodness-of-fit measures for the models. The usefulness of these scores is mixed, given that there are only two data points to define each score. PROP$_3$ and Actransfer errors are very close. This is highlighted by the fact that Actransfer gets a lower MAE for the training

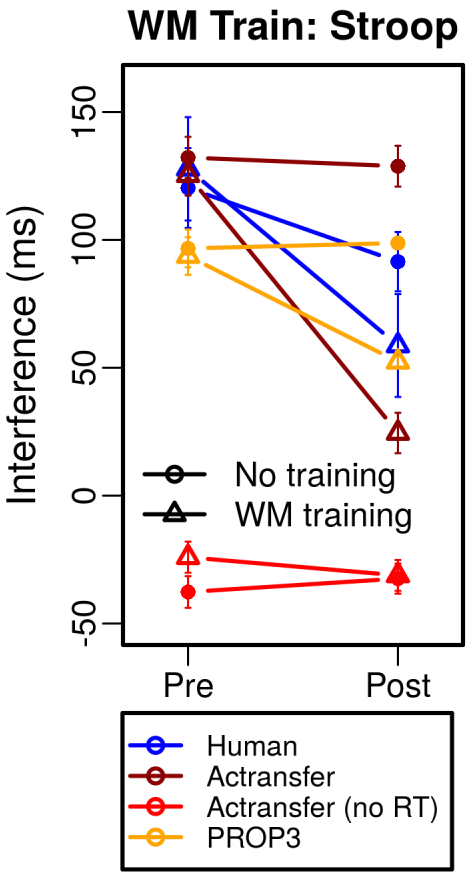


Figure 9.8: PROP$_3$ model interference in the WM/Stroop experiment. Error bars show standard error.

case while PROP$_3$ gets a lower MAPE.

My overall observation from this experiment is that, when I use T_{ps} to model latency for task switching in the manner that would be expected of task sets, I am able to produce the same sort of WM interference effects as Actransfer achieves via $T_{retrieve}$.

	No Training		**WM Training**	
Model	*MAE*	*MAPE*	*MAE*	*MAPE*
Actransfer	24.63	25.4	**18.35**	30.1
Actransfer (No RT)	140.96	133.4	120.89	136.0
PROP$_3$	**15.45**	**13.8**	20.23	**18.8**

Table 9.6: PROP$_3$ goodness-of-fit measures for the WM/Stroop experiment. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

9.4.2.2 Task-switching Experiment

The task-switching experiment from Karbach and Kray (2009) was described in section 5.4 on page 49, but I will review it again here. The experiment is conceptually very similar to the WM/Stroop experiment. Subjects are given a battery of tests before and after a training activity. In this case, the training activity is a task-switching task, and the battery of tests includes a different task-switching task as well as the Stroop task.

In the training task, subjects are iteratively shown different images containing either one or two planes or cars. Subjects have to switch between two different response tasks, given the same types of images. The first task is to report whether the image shows planes or cars. The second is to report whether there are one or two items shown in the image. Subjects train in blocks of just one of these two tasks and in blocks of switching between these two tasks every second trial. Training takes place over the course of four days, each of which involves 8 single-task and 12 task switching blocks. Control subjects, however, train only in single-task blocks.

In the alternate task-switching task that is used as a test before and after training, pictures are of vegetables and fruit rather than cars and planes, and subjects report whether these are small or large instead of whether there are one or two. The Stroop task used as another test is much the same as that used for the WM training experiment described earlier, except neutral trials are used in place of congruent trials (non-color text is used with each font color).

Recall that in the original experiment, humans exhibited transfer to both test tasks when they trained in task-switching blocks. Additionally, while the Stroop interference was better after training in task-switching blocks, it was *worse* after training in only single-task blocks.

Recall also that the Actransfer model design is essentially the same as for the WM training experiment. During training, the agent always practices the "prepare" operation between trials, while during testing it has the choice to either prepare or be idle. Practicing the prepare operation biases the agent to prepare during testing. Training the single-task case in the control setting, by contrast, forces the agent to practice not preparing, which biases the agent to not prepare during subsequent Stroop testing.

I designed the PROP_3 model to also function in the same manner as I did for the WM training experiment. The procedure context hierarchy for this Stroop task is the same as that shown in Figure 9.7. Figure 9.9 shows the procedure context hierarchy for the task-switching tasks. The only difference between the hierarchy for the task-switching training and test is that the test includes instructions for "food-task" and "size-task", which get the food type and size from the prompt, rather than instructions that would get the vehicle type or quantity.

As shown in the figure, the agent has the choice of either `prepare` or `idle` when in between prompts. These determine whether the agent checks which of the two tasks it needs to do next before the next prompt appears or whether it waits to determine this until after the prompt ap-

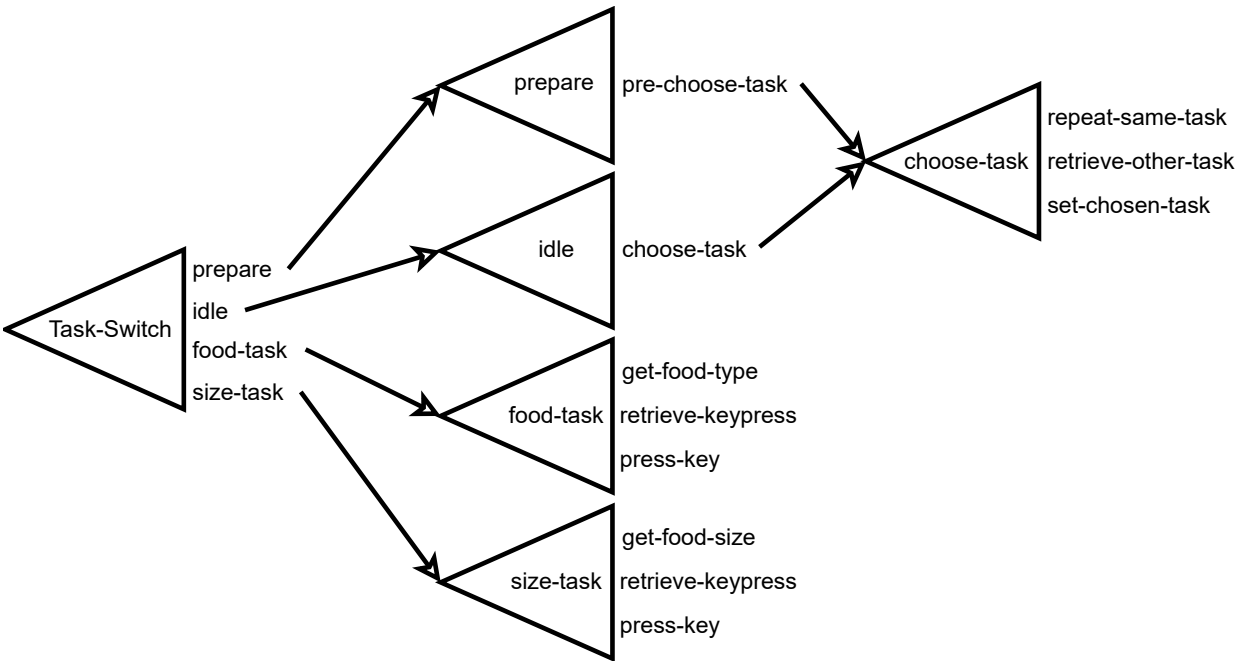


Figure 9.9: The hierarchical goal design of the $PROP_3$ model for the task-switching test task.

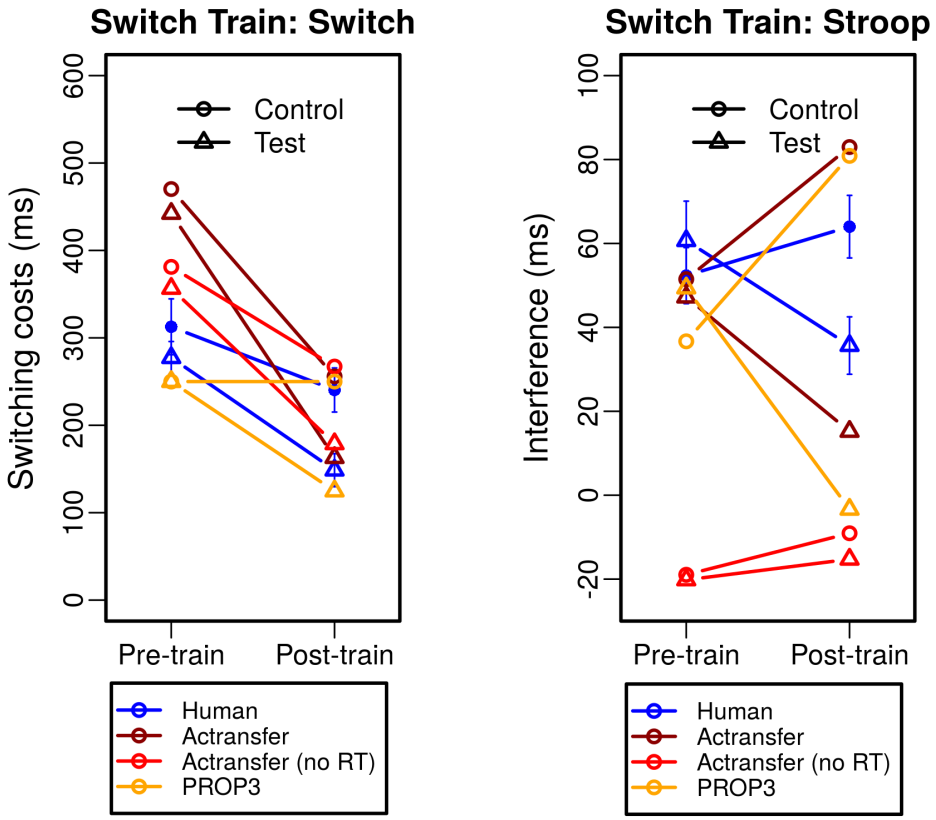
pears. The agent is supposed to switch tasks every second trial. If it selects `prepare` in between prompts, it will immediately then select `pre-choose-task` to retrieve the "choose-task" procedure context. These let the agent reason over whether it needs to either repeat the previous task again or switch to the other task. Once a prompt appears, it can then immediately begin the chosen task, in this case either `food-task` or `size-task`. However, if the agent instead selects `idle`, it will not select `choose-task` until a prompt appears. Only then will it check which task it needs to perform, and then select either `food-task` or `size-task`. This increases the delay between when the prompt appears and when the agent is able to respond.

The procedure contexts for the single-task blocks are the same, except that the agent always invokes the `idle` operator. This is because it does not need to reason over whether it needs to switch tasks.

The Actransfer model for the task-switching tasks adds 0.2 sec for each motor action of pressing a key to answer a prompt, and also adds 0.13 sec for each cognitive action of focusing on the correct answer. The motor/visual times for this Stroop task are the same as for the Stroop task in the previous experiment. Again, $PROP_3$ uses the same times for motor actions but does not add additional time for finding the correct answer.

Figure 9.10a shows the $PROP_3$ model results for the task-switching test task before and after training, alongside both Actransfer and human data. Figure 9.10b similarly shows the $PROP_3$ model results for the Stroop test task before and after training.

In Figure 9.10a, there are a few particularly notable results. First is that the $PROP_3$ model



(a) (b)

Figure 9.10: Human, Actransfer, and PROP$_3$ model results for the task-switching experiment.

switching costs are remarkably close to those of humans compared to the Actransfer agent. Second, however, is that, similar to the WM/Stroop experiment, the PROP$_3$ agent does not show any improvement in the control case after single-task training, where humans do show slight improvement. This is because the PROP$_3$ agent gets little benefit from decision making practice when the agent only has the decision making choice of `idle`. One can also notice that the difference between Actransfer and Actransfer `(no RT)` results is about the same as the difference between Actransfer `(no RT)` and PROP$_3$ results. This indicates that adding the same $T_{retrieve}$ time from Actransfer to PROP$_3$ would result in behavior very close to that of Actransfer `(no RT)`.

In Figure 9.10b for the Stroop test, there are also two notable results. First, the PROP$_3$ agent shows a more extreme reduction in interference than either humans or Actransfer in the test case, though its final interference is close to Actransfer's in the control case. Second, the PROP$_3$ agent shows the same crossover behavior as humans, in which the test subjects begin with more interference than control subjects but end with less. This is difficult to explain as anything other than a random perturbation of the learning algorithm, because the conditions for the agents are identical before training, and the test conditions are also identical. The same can be said of the human data.

Table 9.7 and Table 9.8 show the goodness-of-fit measures for the two test tasks in the task-switching experiment. The $PROP_3$ has the lowest error for the task-switching task, but Actransfer is closer to human performance in the Stroop task. This matches the qualitative behavior of the figures above as discussed.

Switch	**No Training**		**WM Training**	
Model	*MAE*	*MAPE*	*MAE*	*MAPE*
Actransfer	86.55	28.4	89.69	34.6
Actransfer (No RT)	47.69	16.6	54.70	24.4
$PROP_3$	**36.17**	**12.0**	**25.59**	**13.0**

Table 9.7: $PROP_3$ goodness-of-fit measures for the task-switching task. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

Stroop	**No Training**		**WM Training**	
Model	*MAE*	*MAPE*	*MAE*	*MAPE*
Actransfer	**8.82**	**13.9**	**21.39**	**47.1**
Actransfer (No RT)	77.74	133.9	60.33	128.9
$PROP_3$	15.13	26.4	29.62	71.4

Table 9.8: $PROP_3$ goodness-of-fit measures for the Stroop task in the task-switching experiment. MAE is Mean Absolute Error. MAPE is Mean Absolute Percentage Error.

These experiments demonstrate that PRIMs theory combines with Soar theory to produce a novel cognitive architecture model of human task sets that can reproduce task switching, decision making transfer, and WM interference behaviors for these tasks. $PROP_3$ does not use $T_{retrieve}$, and instead models task switching through the time required to change goals in serial with other task operations.

9.5 Discussion

I introduced the $PROP_3$ model as a unification of PRIMs theory and Soar theory that is consistent with both theories and satisfies my desiderata for this thesis, as depicted in Figure 9.11. $PROP_3$ addresses the implementation of P3 and P6, the final gaps I identified in chapter 4. I discovered that the resulting procedure contexts model aligns with task set theory. When I used the T_{ps} latency of accessing procedure contexts to model WM interference and task-switching costs in a manner consistent with task set theory, I were able to replicate Actransfer and human performance trends in the final two experiments from the Actransfer suite in a task-independent manner.

Though $PROP_3$ satisfies my desiderata as a comprehensive, runnable, task-independent, and consistent model of PRIMs procedural learning, it is more limited than $PROP_2$ or Actransfer in

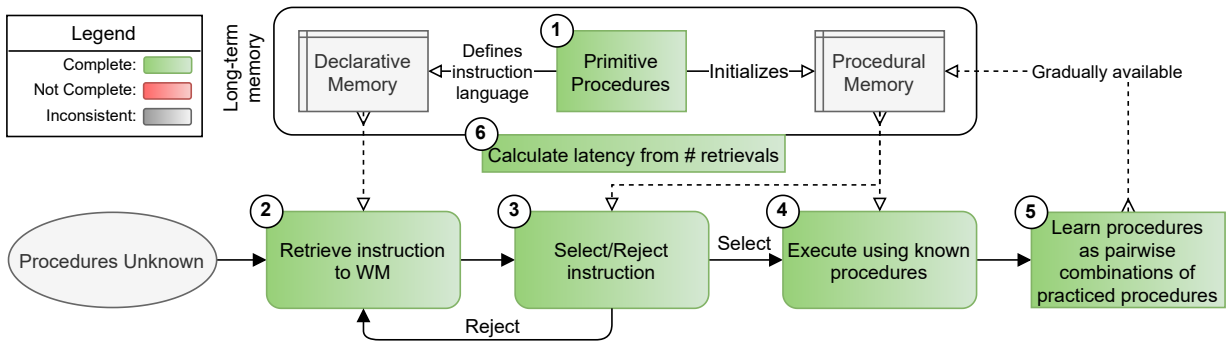


Figure 9.11: The PROP_3 flow diagram and completeness for my desiderata.

its ability to replicate some human behaviors, particularly in the editors task. The PROP_2 model was able to use the same $T_{retrieve}$ timing as Actransfer with a novel algorithm for cognitive phase learning to achieve a closer fit to human performance in the editors task, but the computation of this model was not consistent with the architectural theory of Soar, nor did it easily allow a solution for P2/P3 problem in which P2 only ever selects a single instruction for P3. This implies that the high-level approach of PROP_2 for cognitive phase learning has merit, but that its implementation needs further work.

Is it possible for P2 to select multiple instruction choices for P3 at once while also using the spreading activation approach of PROP_2? This would still require retrieving multiple instructions at once, and for Soar this would still necessitate the procedure context structure or something very similar that is based on Soar problem spaces. In order for P2 retrievals to still be primarily based on spreading activation, P3 decisions could only lead to an increased bias for retrieving the selected problem space instructions, and still leave the actual procedure context retrieval up to activation. It makes sense that there should be some manner of spreading activation for retrieving procedure contexts. When an unexpected task prompt appears, unconnected with any structures currently in WM, the agent would need a spontaneous or spreading-based retrieval in order to adapt to that new prompt. For tasks such as those used in the Actransfer experiment suite, this kind of unexpected stimulus behavior is not relevant. I discuss this question further in appendix sections C.2.2-C.2.3.

My evaluation of the completed PROP_3 model shows that keeping the constraints of my desiderata limits what one can model with PRIMs processing in a cognitive architecture like Soar. It also shows that keeping these constraints can lead to new insights such as the connection between PRIMs theory and task set theory as well as the novel model of task-independent value function features for RL.

In the first evaluation, the PROP_3 models provided a nearly exact fit to human performance in the arithmetic task training, but otherwise performed comparably to the Actransfer (`No RT`) agent. In the second evaluation, the PROP_3 model replicated the same human learning behaviors as Ac-

transfer without relying on $T_{retrieve}$. PROP$_3$ performance in WM interference and switching costs was slightly low overall compared to human performance. A process of speeding up declarative retrievals with increased activation could potentially account for this deficit. However, since these experiments measure performance differences, any additional sources of latency would need to affect agent processing unequally across the difference components in order to affect model results. As it stands, the PROP$_3$ model as a comprehensive model of PRIMs theory provides comparable fits to human data as Actransfer while supplying more computational detail.

More detailed experimentation is desired to determine the practical effects of RL parameters in PROP$_3$ for any task. However, the purpose of experimentation for this thesis is to evaluate the theoretical and computational changes introduced with PROP$_3$, not to analyze the parameters of reinforcement learning systems. The PROP$_3$ system does not attempt to constrain how RL parameters are used in Soar any more than Soar itself does.

It should be acknowledged that the model behavior for the second evaluation, whether using Actransfer or PROP$_3$, is highly sensitive to changes in the model parameters and to the particulars of the agent reasoning, particularly for the task-switching experiment Stroop task. It is possible to get very different model behaviors by changing the environment reward or by changing how much rule knowledge the agent already knows at the start of its tasks. For the purposes of this thesis, results with PROP$_3$ merely demonstrate that the procedure context approach is capable of reproducing equivalent behaviors as published for Actransfer for these experiments.

T_{ps} includes the decision cycles used to retrieve a procedure context. Procedure contexts are not meant to model generic declarative facts, and a different approach for timing classic declarative fact retrievals would be desirable. Rather, procedure contexts are merely meant to correspond with the specific kind of WM knowledge described by task set theory, sometimes called procedural WM (Oberauer, 2010). It is likely that Soar would need to be modified to simulate activation-based retrieval latency to model declarative learning to the extent that ACT-R does. However, as Oberauer discusses, it seems that the human mechanic for declarative memory access is similar, though distinct, to the task set mechanic for procedural memory access. If procedure contexts have merit as a model of task sets, it is possible that a similar computational approach could be used to explain declarative memory access in a more detailed manner than $T_{retrieve}$ does. Unless further research should show otherwise, however, T_{ps} should not be considered a replacement for activation-based declarative retrieval latency such as ACT-R's $T_{retrieve}$. Rather, problem-space latency should be considered an additional factor that has been previously overlooked in cognitive modeling.

An important aspect of T_{ps} is that it does not predict faster performance the more a memory is accessed the way that activation-based latency does. A PROP$_3$ agent could only reduce T_{ps} by reducing the number of cycles that it must use to access a memory. One way to do this would be by

rearranging procedure context links into more compact hierarchy. Another way might be through some sort of short-term caching of recently-used memories. Still another way is spontaneous retrieval of relevant memories in the manner discussed in appendix section C.2.2.

Procedure contexts are theoretically applicable in other architectures wherever declarative instructions can be clustered by task contexts or goals. Yet their use would be cumbersome without the ability to evaluate condition lines in parallel as I do using Soar elaboration rules. Thus, while ACT-R has used goal stacks in the past, ACT-R lacks the ability to fire multiple rules per decision cycle, and this would make it very difficult if not impossible to use procedure contexts in ACT-R.

9.6 Soar Agent Design

The PROPs system agent code, completed with $PROP_3$, represents a task-independent basis for future cognitive modeling in Soar. I stated in chapter 1 that this represents a secondary contribution of this thesis.

As discussed in relation to my methodology in chapter 3, PROPs was built as an agent rather than an architectural modification in order to both simplify and constrain its iterative development. However, this also has the benefit of making PROPs easy to use by other researchers. Because the agent code is task-independent, a researcher can load it as a library within their own agent design while using the same publicly available version of Soar.

My modifications to the Soar architecture that make chunking gradual for $PROP_2$ and $PROP_3$ are not, however, a part of the main distribution of Soar 9.6.[4] The PROPs system code will not generate the same gradual learning behavior as my experiments did here without this architectural modification. The code will function, but it will chunk rules in a one-shot manner, as Soar normally does. This will not impair the $PROP_3$ agent capabilities in any way other than to prevent it from taking more time, since a Soar model of PRIMs theory such as $PROP_3$ is not able to increase its transfer due to gradual chunking.

Though PROPs was built as a cognitive model, in its final form it is fairly fast compared to many models. Even when using gradual chunking, it is able to operate at about 30-40 times real time for the human experiments described in this thesis, depending on the environment. Any agents developed using the PROPs system will have the advantage of the efficiency of Soar as an artificial intelligence platform with less development effort due to the generality with which PROPs defines how Soar mechanisms interact. Combined with the fact that programming a PROPs agent requires less effort compared to a Soar agent, since the PROPs system code provides infrastructure for managing goals and memory systems, PROPs could potentially be useful as an artificial intelligence

[4]The architecture modification is publicly available, however, within the open-source Soar repository. It can be found at: https://github.com/SoarGroup/Soar/tree/cbc-experimental

tool as well.

CHAPTER 10

Discussion and Related Work

In this chapter, I discuss some of overall theory related to PRIMs and the PROPs system as well as some related work that appears to be connected with the results I described above.

I revisit the identifiability problems mentioned at the beginning of this thesis. I then discuss the question of researching primitive learning using adult learners, followed by an exploration of the questions of cognitive phase learning for PRIMs and of goal stacks in cognitive architecture theory. I conclude with a discussion of related work in rapid task learning and task switching.

Appendix C contains further discussion on the various implications that follow from the technical details of the completed PROPs system. See section C.5.2 in particular for discussion of how neither Soar nor ACT-R currently supports all the computation that this research implies for PRIMs theory.

10.1 Identifiability Problems

In chapter 2, I discussed the identifiability problems in cognitive modeling, in which models that function differently can still equally capture human task behaviors. This makes it difficult to determine when a model is truly an accurate model of human cognition.

The PROPs system and model that I introduce in this thesis provided several novel insights and approaches for cognitive modeling, such as PRIM resolution, learned P2 in $PROP_2$, and procedure contexts in $PROP_3$. Taken individually, these are still ways of producing similar behavior as Actransfer by different means, and there are still identifiability issues. However, this work shows how progress can be made with identifiability when a single modeling system is constrained by the four desiderata I use for the PROPs system. When I demand with D3 that the model is reusable across several different tasks and domains, each additional task modeled reduces the search space of possible models. For instance, when I require the same P6 for both the arithmetic and editors tasks, I reveal where PRIMs processing comes short in modeling human learning in the editors task. Any future work that attempts to model the editors task with PRIMs can leverage this result

in an attempt to identify the missing timing factors for that task. And when I require with D4 that the model be consistent as a single architectural system and model of human cognition, I identify how different model approaches come up short in their support for each other. Further, when I constrain one process to be consistent with another, this cuts down the search space for possible models and presents a single solution for testing. For instance, when I require that the approach of P2 support Soar's decision making theory for P3, this not only shows how $PROP_2$'s solution for P2 comes up short but also directly leads to the procedure context model as a solution.

Thus, I show that the constraints of my desiderata are powerful tools for cognitive modeling. Testing a modeling approach according with these constraints does require an expansive single system, such as the PROPs system described in this thesis. For instance, I would not have been able to easily constrain the consistency of my approach for P2 without the broader setting of P1-P6 or the architectural basis of Soar. This requires a greater long-term effort and commitment when developing a novel cognitive model. However, a takeaway of this thesis is that this approach can yield true progress in combating the identifiability problems.

10.2 Adult Learning with Primitives

One potential question is whether it makes sense to model *adult* human learning by composing computational primitives in the manner described by PRIMs theory and as Taatgen (2013) did with the Actransfer experiments. Should not the adult participants in the original human studies have already composed their task-independent primitives into more complex procedures before they began those tasks? Would adults still be directly invoking primitive procedures at all? These are very important questions when using PRIMs theory and when evaluating its experimental results.

Before considering an answer, the fact remains that the more primitive, task-independent layer of procedure introduced by PRIMs theory does allow the Actransfer and PROPs agents a greater level of transfer, which appears similar to that displayed by humans. The real theoretical question, however, is what exactly would these deeper primitive elements correspond with in human learning? If PRIM theory really does represent an accurate model of adult human learning, there are three possible answers that are consistent with PRIMs computation.

One answer is that the PRIM operation types might provide an accurate human model but that the number of ways of resolving these operations to specific WM elements in humans is so vast that even an adult human would not meaningfully transfer prior life experience to these tasks. In other words, the PRIM operations would be specific to each task. This seems unlikely at a surface level, given that the kinds of operations performed in these tasks are not so unusual that one would expect humans to have never attempted anything similar before. And part of the point of PRIMs is that they are highly transferable.

However, if mental representations for tasks can drift over longer periods of time, then procedural learning for tasks from earlier in life might not be as transferable a long time later, even if the procedures were fairly permanent knowledge. A subject might have similar representations between a task learned today and a task learned yesterday, but have a dissimilar representation for a task learned over a year ago, even if that older task was externally very similar to a task from today. This would imply more transfer of recent procedural knowledge and less of older knowledge. This assumes that mental representations for tasks are not particularly constrained, which is true in Soar.

Another possibility is that the kind of compositional learning done with PRIMs does not actually result in permanent procedural knowledge, but knowledge that endures at a shorter scale, perhaps for only a few months. However, true human procedural knowledge is still considered a fairly permanent memory store (Radvansky & Tamplin, 2012), so this seems less likely. However, some mental skills are known to only last on the order of months, or to get replaced when trained in incompatible skills, such as was shown in the task-switching experiment when subjects had negative transfer from single-task practice to the Stroop task (Karbach & Kray, 2009). It could be that the hierarchy of PRIMs compositions represents human knowledge that can similarly be altered over time for either positive or negative transfer, depending on where it is applied.

A final possibility, though, is that, while the main principles of PRIM modeling might be accurate, the complexity of the cognitive operations needed for these tasks is much greater than is reflected in the task instructions of the Actransfer models. The Actransfer model instructions used no more than two or three condition and action lines per instruction, on average. It is quite conceivable that adult humans would actually perform a vastly greater number of condition evaluations and action operations per decision cycle. In practice the model might capture this behavior as only a small handful of condition and action lines because adult humans are able to abstract a great number of their operations into a relatively small handful of grouped condition and action line steps, due to prior learning that they have built up by the time they are adults. This would imply that while the PRIM operation types might be accurate, the WM space in which they can be applied is much more complex and requires a much deeper composition hierarchy than would be apparent from the models I discussed in this work.

10.3 Procedure Comprehension

As stated in chapter 2, Bovair and Kieras (1991) observed a distinction in procedural learning between procedure comprehension and procedure interpretation. Procedure comprehension corresponds with cognitive phase learning and what in PRIMs theory would be the process of building task instructions based on environment interaction. Procedure interpretation uses those instructions

to perform task skills.

PRIMs theory does not address where task instructions come from and assumes that some prior procedure comprehension process or cognitive phase learning has created these in LTDM. Research in procedure comprehension can to some degree illuminate some of the properties of task instructions that PRIMs processing would have to use.

Bovair (1992) discusses several lines of research that imply that human instruction representations are hierarchical internally, and that it is helpful for human learners when task directions are given in the form of explicit, hierarchical goals. This supports the PROPs procedure context model that represents task instructions with a hierarchical structure that also corresponds with a hierarchy of subgoals.

Beydoun and Hoffmann (2001) extends this study and argues for a model of skill knowledge representation called the Nested Ripple Down Rules (NRDR) framework. This framework arranges rule knowledge hierarchically in a branching tree of conditions very similar to my model of elaboration contexts, but with a different approach that nests more constrained and discriminatory rule representations under more general rules. PROPs, by contrast, nests elaboration contexts if they are component processes of parent contexts rather than if their rules have similar structure. Future work with PRIMs theory might explore using this framework for representing hierarchical task instructions and test it in the context of Soar or other cognitive architectures.

10.4 Theory of Goal-Stacks

The procedure context is closely connected with Soar's PSCM theory and its notion that the architecture automatically manages a stack of goals. I use the Soar goal stack as the foundation for my representation of active task sets. One potential question is whether goal stacks are plausible in human modeling. While architecturally-supported goal stacks were widely supported in cognitive modeling in both Soar and ACT-R, researchers using ACT-R in the early 2000s determined that goals were the same as generic declarative facts in human cognition, and that the architecture should therefore not treat goals differently (Anderson & Douglass, 2001). The goal stack as an architecturally-maintained construct was removed from ACT-R shortly thereafter. However, I observe a distinction between the concept of "goal" that Anderson and Douglass measured and the concept of goal as used in a goal stack such as Soar's.

In their experimentation, Anderson and Douglass modeled a "goal" as an *image of a desired task action*. They experimented using the classic Towers of Hanoi puzzle, in which subjects move disks around various pegs to transition from a starting disk configuration to a desired end configuration. Subjects would, theoretically, imagine a "goal" of moving a disk from one location to another. If that movement was blocked by another disk currently in the destination location,

subjects could push onto their goal-stack a new "goal" of removing the blocking disk from the destination. Thus, the goal-stack represented a stack of temporally-sequenced planned actions. They measured that, with humans, the time it took to retrieve knowledge of the next action in the plan improved with practice in the same manner that normal declarative retrievals could get faster with practice. Thus, they influentially concluded,

> "ACT–R and Soar are wrong in their assumption of a special goal stack. Goals appear to behave like any other memory objects. Goals set in the process of subgoaling are probably no different than other sorts of intentions that people set."

I argue that their results do not, in fact, oppose the PROP_3 model and use of modern Soar's "goal stack". While a Soar (or formerly ACT-R) goal stack can be used as an action-plan stack, it does not have to be used that way. Fundamentally, Soar's stack is a stack of WM states, which each correspond with a goal and problem space. PROP_3 demonstrates that this stack makes sense in cognitive modeling when the states in the stack correspond with problem spaces rather than planned actions. A stack of strategically-planned actions is an explicit form of declarative knowledge, comparable to a stack of memorized numbers. One would expect a stack of planned actions to have the appearance of a normal declarative memory structure. But the Soar state stack is explicitly designed to support problem-spaces, based on Soar's PSCM theory. A problem space in Soar is not an explicit plan, but rather it is an implicit group of operators that might be applied toward a goal in a state's context.

10.5 Rapid Task Switching

Somewhat related to the question of goal representation in the PROPs system is the question of whether and how an agent might autonomously pursue multiple fairly unrelated goals at once. For example, consider a chef who must simultaneously attend a boiling pot of pasta on a stove, mix cake batter, and cook fish in an oven. It is up to the chef to determine when to switch among managing each of the stove, mixer, and oven in such a way that uses available time and resources efficiently. Salvucci and Taatgen (2008) have proposed a model of rapid task switching in which a cognitive architecture lets an agent pursue multiple goals each with a separate "thread" of cognitive processing. (See discussion by Lui and Wong (2020) for how this and related theories of multitasking relate to recent research on task sets.) In this model, each thread independently competes for limited cognitive resources such as declarative memory, procedural memory, vision, or motor processing in pursuit of that thread's associated goal, and when a thread completes a segment of work with a resource it releases that resource for other threads to use. Salvucci and Taatgen implement their model with a modification of the ACT-R architecture that allows multiple goals in the

goal buffer. With each decision cycle the agent can fire a rule that matches any one of the goals in the goal buffer so long as the architecture modules required by that rule are available. A functional benefit of this approach is that the agent can compose multiple competing tasks arbitrarily on-line and does not require an explicit hierarchy of tasks in LTDM.

As described in chapter 9 for the WM/Stroop and task-switching experiments, a PROPs agent supports deliberate task-switching via a top-level elaboration context in WM that references each of the different tasks the agent can switch among. The agent uses this top-level context to propose an operator for each possible competing task or goal. The agent can select different subtasks either in response to direct task prompts or based on preference rules in some other conditioned manner. But when would the agent assemble the top-level elaboration context that would allow competing goals? PROPs does not attempt to explain the formation of procedure context hierarchies in LTDM, but the question of goal representation is still important. Assuming the PROPs agent does not have a memorized tree of procedure contexts already in LTDM for each collection of specific tasks it must switch among, which in the chef example for instance seems unlikely, then we can assert that somehow the PROPs agent would have to be able to create this elaboration context on-line in WM.

Future work with PROPs might explore adding the ability for the agent to iteratively retrieve multiple different procedure contexts from LTDM and rearrange these into a single short-term procedure context in its WM. In this design, an agent would be able to perform a procedure context retrieval in a WM substate even when it already has a procedure context loaded in that substate. It would then need to either merge the additional retrieved context into its existing context or ignore it. Note that this would also affect T_{ps}, because the agent would not need to spend a cycle to create a substate before retrieving an additional context. In this case, incorporating additional procedures into an already-active context would require only 100 msec by my current modeling approach, for the two cycles needed to retrieve a context, rather than 150 msec.

If a PROPs agent were able to compile an elaboration context on-line that would mediate its multitasking, it could benefit in future attempts at multitasking if it were additionally able to store that elaboration context in its LTDM. Alternately, an agent might learn to reduce the different operator instructions from multiple elaboration contexts into a single large context. In this case, the agent would not even need to enter a substate to access different actions for subtasks, but would effectively represent the collection of subtasks as a single task. If an agent were able to compile multiple task instructions together into a single procedure context in this manner, this would remove task switch costs. Internally, the agent would represent the multiple subtasks as if they were a single task, and so no switching would be required.

10.6 Rapid Instruction Task Learning

If future work should attempt to extend the PROPs system with the ability to create or modify procedure contexts on-line, to include a model of cognitive phase learning, it could draw heavily from existing research in Rapid Instructed Task Learning (RITL) (Cole et al., 2013). RITL is learning a new rule or task rapidly from given instructions, often as quickly as one-shot learning. It stands in contrast to the kind of gradual learning often seen in RL or sustained human practice, where successfully completing a task can require tens, hundreds, or thousands of repeated practice attempts.

Cole and colleagues discuss extensive work in neuroscience and modeling that attempts to understand the human ability to parse task instructions into a task set form that lets one immediately perform the instructed behaviors. There are many similarities between their RITL view of task instructions and task sets and the implications of the completed PROPs model, from the combinatoric nature of composing rule primitives, hierarchical transfer, variable binding for groups of neurons, hierarchical growth from procedural primitives, and the need for declarative reasoning to control task set formation. There is also work observing that RITL serves as the first of three phases of learning, (1) RITL, (2) controlled, and (3) automatic, very similar to the three phases of Fitts and Posner (Chein & Schneider, 2012).

As mentioned in section 7.5, by looking at the neural mechanisms of RITL, Cole and colleagues also predict that gradual learning might be useful for building more transferable hierarchical representations for tasks. In the context of $PROP_3$, RITL is likely be the process of learning elaboration contexts rather than apply contexts. Apply contexts in the PROPs model represent the hierarchy of gradually-learned procedural skill that are composed largely in the associative phase, not the cognitive/RITL phase. Future work could investigate potential neural correlates of apply contexts further and whether these too might align with theory for task sets or whether these might map more accurately to structures in other regions of the brain, such as perhaps the pre-motor or motor cortex.

CHAPTER 11

Conclusion

This thesis presented a comprehensive computational model of PRIMs theory for task-independent human procedural learning. The PROPs system satisfies my four desiderata outlined in chapter 1.

D1. **The model is comprehensive.** That is, it specifies and constrains the cognitive processing details that are necessary to execute each of the six phases of PRIMs theory.

D2. **The model is able to perform its tasks.** That is, it is implemented within the Soar cognitive architecture's input/output environment and has thereby been demonstrated to replicate human learning behavior in practice.

D3. **The model is task-independent.** That is, it is implemented as a single system that is the same for all experimental tasks, with variation in the form of different task instructions given to the system.

D4. **The model is a model of human information processing.** That is, the model's computation is informed by and consistent with the current understanding of the human cognitive architecture and various lines of human research, particularly three-phase theory and task set theory.

In the process of satisfying these desiderata, I introduced four main theoretical contributions to the field of cognitive architecture research for task-independent procedural learning.

- ✠ **I introduced PRIM resolution as a deeper layer of primitive procedural memory processing to the production system architecture model of procedural learning.** I showed in section 7.7 that this can provide a more human-like power-law learning and transfer profile absent from Actransfer results.

- ✠ **I introduced a new approach by which a cognitive model can learn to use the state of working memory to guide long-term declarative memory retrievals.** I showed in section

8.4 that this learning process replicates cognitive phase learning in ways that previous PRIMs models could not.

- ✠ **I extended the Soar cognitive architecture to support gradual procedural learning in a manner identified to be consistent with existing architecture theory.** I showed in sections 8.4 and 9.4 that this can be used to replicate the gradual human learning desired in PRIMs theory.

- ✠ **I introduced a novel cognitive architecture model of human task sets in the form of procedure contexts.** I showed in section 9.4 how this can be used to model human choice-based decision making, task switching, and working memory interference effects.

In addition to these, the PROPs agent code provides a task-independent software basis that could be used for rapidly developing future cognitive models with Soar. The methodology I demonstrated with this thesis can also be applied in the future to compare other architectures and cognitive processing theories and to develop new and greater advances within the field of cognitive architecture.

Future work can branch out in several directions from this thesis. One might explore further developments to either the Soar or ACT-R cognitive architectures to better incorporate the kinds of automatic processing supported by this research. Actransfer or its experimental descendant architectures might incorporate some form of PRIM resolution. Soar might be modified to create a time-based trace of behavior that includes time for each declarative retrieval. In general, one might also explore what kinds of modifications would be necessary for Soar to support the processing of $PROP_2$ in a manner that satisfies my desiderata.

Another direction for future research is that of expanding the PROPs model to tackle the problem of declarative reasoning and cognitive phase learning. A truly comprehensive and task-independent model of human reasoning that satisfies all my desiderata would be a difficult accomplishment. A more tangible goal that works in this direction is to expand the $PROP_3$ model so that the agent can modify its procedure context structures on-line to produce cognitive phase learning behavior in a manner consistent with the findings from $PROP_2$. Such work might draw inspiration from Bovair (1992)'s model of procedure comprehension processing.

Finally, the findings from my work with $PROP_3$ imply connections with neuroscience theory related to task sets and RITL. Future work should investigate whether those lines of research imply additional ways to extend the PROPs model. Such an investigation might also incorporate Oberauer (2010)'s theory of procedural and declarative WM to extend $PROP_3$ procedure contexts as task sets into similar computation for declarative memory sets.

Bibliography

Anderson, J. R., Betts, S., Bothell, D., Hope, R., & Lebiere, C. (2019). Learning rapid and precise skills. *Psychological Review*, *126*(5), 727–760.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, *89*(4), 369–406.

Anderson, J. R. (1987). *Language, memory, and thought*. Lawrence Erlbaum.

Anderson, J. R. (1993). *Rules of the mind*. L. Erlbaum Associates.

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

Anderson, J. R., & Douglass, S. (2001). Tower of Hanoi: Evidence for the cost of goal retrieval. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *27*(6), 1331–1346.

Arslan, B., Verbrugge, R., & Taatgen, N. (2017). Cognitive control explains the mutual transfer between dimensional change card sorting and first-order false belief understanding: A computational modeling study on transfer of skills. *Biologically Inspired Cognitive Architectures*, *20*, 10–20.

Beck, J. E., & Chang, K.-m. (2007). Identifiability: A fundamental problem of student modeling. In C. Conati, K. McCoy, & G. Paliouras (Eds.), *User modeling 2007* (pp. 137–146). Springer Berlin Heidelberg.

Beydoun, G., & Hoffmann, A. (2001). Theoretical basis for hierarchical incremental knowledge acquisition. *International Journal of Human-Computer Studies*, *54*(3), 407–452. https://doi.org/https://doi.org/10.1006/ijhc.2000.0445

Bovair, S. (1992). *A model of procedure acquisition from written instructions*. University of Michigan. https://deepblue.lib.umich.edu/handle/2027.42/105854

Bovair, S., & Kieras, D. E. (1986). The acquisition of procedures from text: A production-system analysis of transfer of training. *Journal of Memory and Language*, *25*(5), 507–524. https://doi.org/10.1016/0749-596X(86)90008-2

Bovair, S., & Kieras, D. E. (1991). Toward a model of acquiring procedures from text. *Handbook of reading research*, *2*, 206–229.

Brasoveanu, A. (2015). *Intro to the ACT-R subsymbolic level for declarative memory*. https://people.ucsc.edu/~abrsvn/ACT-R_subsymbolic_3.pdf

Chein, J. M., & Morrison, A. B. (2010). Expanding the mind's workspace: Training and transfer effects with a complex working memory span task. *Psychonomic Bulletin & Review*, *17*(2), 193–199.

Chein, J. M., & Schneider, W. (2012). The brain's learning and control architecture. *Current Directions in Psychological Science*, *21*(2), 78–84. https://doi.org/10.1177/0963721411434977

Cole, M. W., Laurent, P., & Stocco, A. (2013). Rapid instructed task learning: A new window into the human brain's unique capacity for flexible cognitive control. *Cognitive, Affective, and Behavioral Neuroscience*, *13*, 1–22. https://doi.org/10.3758/s13415-012-0125-7

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, *24*(1), 87–114. https://doi.org/https://doi.org/10.1017/S0140525X01003922

Cowan, N. (2010). The magical mystery four: How is working memory capacity limited, and why? *Current directions in psychological science*, *19*(1), 51–57. https://doi.org/https://doi.org/10.1177/0963721409359277

Cowan, N. (2017). The many faces of working memory and short-term storage. *Psychonomic Bulletin & Review*, *24*(4), 1158–1170.

Dosenbach, N. U., Visscher, K. M., Palmer, E. D., Miezin, F. M., Wenger, K. K., Kang, H. C., Burgund, E. D., Grimes, A. L., Schlaggar, B. L., & Petersen, S. E. (2006). A core system for the implementation of task sets. *Neuron*, *50*(5), 799–812.

Dreisbach, G., & Haider, H. (2008). That's what task sets are for: Shielding against irrelevant information. *Psychological Research*, *72*(4), 355–361.

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

Eriksson, J., Vogel, E. K., Lansner, A., Bergström, F., & Nyberg, L. (2015). Neurocognitive architecture of working memory. *Neuron*, *88*(1), 33–46.

Faghihi, U., & Franklin, S. (2012). The LIDA model as a foundational architecture for AGI. In P. Wang & B. Goertzel (Eds.), *Theoretical foundations of artificial general intelligence* (pp. 103–121). Atlantis Press. https://doi.org/10.2991/978-94-91216-62-6_7

Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, *47*(6), 381–391. https://doi.org/10.1037/h0055392

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

Gray, W. D., & Lindstedt, J. K. (2017). Plateaus, dips, and leaps: Where to look for inventions and discoveries during skilled performance. *Cognitive Science*, *41*(7), 1838–1870. https://doi.org/10.1111/cogs.12412

Huijser, S., van Vugt, M. K., & Taatgen, N. A. (2018). The wandering self: Tracking distracting self-generated thought in a cognitively demanding context. *Consciousness and Cognition*, *58*, 170–185.

Jain, A., Bansal, R., Kumar, A., & Singh, K. (2015). A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students. *Int J App Basic Med Res*, *5*(2), 124–127.

Johansson, C., & Lansner, A. (2007). Towards cortex sized artificial neural systems. *Neural Networks*, *20*(1), 48–61. https://doi.org/https://doi.org/10.1016/j.neunet.2006.05.029

Johnson, T. R. (1997). Control in ACT-R and Soar. *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, 343–348.

Jones, R. M., Lebiere, C., & Crossman, J. A. (2007). Comparing modeling idioms in ACT-R and Soar. *Eighth International Conference on Cognitive Modeling*, 49–54.

Jones, S. J. M., Wandzel, A. R., & Laird, J. E. (2016). Efficient computation of spreading activation using lazy evaluation. *International Conference on Cognitive Modeling*.

Karbach, J., & Kray, J. (2009). How useful is executive control training? age differences in near and far transfer of task-switching training. *Developmental Science*, *12*(6), 978–990. https://doi.org/10.1111/j.1467-7687.2009.00846.x

Kennedy, W. G., & Trafton, J. G. (2007). *Long-term symbolic learning in Soar and ACT-R* (tech. rep.). Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence. 4555 Overlook Avenue SW,Washington,DC,20385.

Kieras, D. E. (2016). A summary of the EPIC cognitive architecture. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780199842193.013.003

Kim, J. W., & Ritter, F. E. (2015). Learning, forgetting, and relearning for keystroke- and mouse-driven tasks: Relearning is important. *Human-Computer Interaction*, *30*, 1–33. https://doi.org/10.1080/07370024.2013.828564

Kim, J. W., Ritter, F. E., & Koubek, R. (2013). An integrated theory for improved skill acquisition and retention in the three stages of learning. *Theoretical Issues in Ergonomics Science*, *14*(1), 22–37. https://doi.org/10.1080/1464536X.2011.573008

Kirk, J. (2019). *Learning hierarchical compositional task definitions through online situated interactive language instruction.* (Doctoral dissertation). University of Michigan. http://hdl.handle.net/2027.42/153434

Kirk, J., & Laird, J. E. (2019). Learning hierarchical symbolic representations to support interactive task learning and knowledge transfer. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 6095–6102.

Kotseruba, I., & Tsotsos, J. K. (2018). 40 years of cognitive architectures: Core cognitive abilities and practical applications. *Artificial Intelligence Review*. https://doi.org/10.1007/s10462-018-9646-y

Kump, B., Moskaliuk, J., Cress, U., & Kimmerle, J. (2015). Cognitive foundations of organizational learning: Re-introducing the distinction between declarative and non-declarative knowledge. *Frontiers in Psychology*, *6*(1489), 1–12. https://doi.org/10.3389/fpsyg.2015.01489

Laird, J. E. (2012). *The Soar cognitive architecture*. MIT Press.

Laird, J. E., Gluck, K., Anderson, J., Forbus, K. D., Jenkins, O. C., Lebiere, C., Salvucci, D., Scheutz, M., Thomaz, A., Trafton, G., Wray, R. E., Mohan, S., & 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.

Li, N. H. (2016). *The goal re-activation problem in cognitive architectures* (Doctoral dissertation). University of Michigan.

Lui, K. F. H., & Wong, A. C.-N. (2020). Multiple processing limitations underlie multitasking costs. *Psychological Research*, *84*(7), 1946–1964. https://doi.org/10.1007/s00426-019-01196-0

Martin, K., Staiano, W., Menaspà, P., Hennessey, T., Marcora, S., Keegan, R., Thompson, K., Martin, D., Halson, S., & Rattray, B. (2016). Superior inhibitory control and resistance to mental fatigue in professional road cyclists. *PloS one*, *11*, e0159907. https://doi.org/10.1371/journal.pone.0159907

Muller, T. J., Heuvelink, A., & Both, F. (2008). Implementing a cognitive model in Soar and ACT-R: A comparison. *Proceedings of the Sixth International Workshop: From Agent Theory to Agent Implementation*.

National Research Council. (1994). Transfer: Training for performance. In D. Druckman & R. A. Bjork (Eds.), *Learning, remembering, believing: Enhancing human performance* (pp. 25–56). The National Academies Press. https://doi.org/10.17226/2303

Newell, A. (1973). You can't play 20 questions with nature and win: Projective comments on the papers of this symposium. *Visual information processing* (pp. 283–308). Elsevier Inc.

Newell, A. (1990). *Unified theories of cognition*. Harvard University Press.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Prentice Hall.

Oberauer, K. (2010). Declarative and procedural working memory: Common principles, common capacity limits? *Psychologica Belgica*, *50*(3-4), 277–308.

Perez, R., Gray, W. D., Posner, M., Vinogradov, S., & Chi, M. (2018). Learning-to-learn from novice to expertise: New challenges and approaches for one of the oldest topics of cognitive science. *Cognitive Science*.

Radvansky, G. A., & Tamplin, A. K. (2012). Memory. In V. S. Ramachandran (Ed.), *Encyclopedia of human behavior (second edition)* (Second Edition, pp. 585–592). Academic Press. https://doi.org/https://doi.org/10.1016/B978-0-12-375000-6.00229-9

Rangelov, D., Töllner, T., Mueller, H., & Zehetleitner, M. (2013). What are task-sets: A single, integrated representation or a collection of multiple control representations? *Frontiers in Human Neuroscience*, *7*(524), 1–11. https://doi.org/10.3389/fnhum.2013.00524

Rice, P., & Stocco, A. (2018). Mechanisms of rule resolution in premotor cortex: A combined TMS/computational modeling study. *16th International Conference on Cognitive Modelling*, 108–113.

Ritter, F. E., Tehranchi, F., & Oury, J. D. (2019). ACT-R: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, *10*(3), e1488.

Ritter, F. E., & Wallach, D. P. (1998). Models of two-person games in ACT-R and SOAR. *Second European Conference on Cognitive Modelling*, 202–203.

Sabah, K., Dolk, T., Meiran, N., & Dreisbach, G. (2019). When less is more: Costs and benefits of varied vs. fixed content and structure in short-term task switching training. *Psychological Research*, *83*, 1532–1542. https://doi.org/10.1007/s00426-018-1006-7

Sakai, K. (2008). Task set and prefrontal cortex [PMID: 18558854]. *Annual Review of Neuroscience*, *31*(1), 219–245. https://doi.org/10.1146/annurev.neuro.31.060407.125642

Salvucci, D. D., & Taatgen, N. A. (2008). Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, *115*(1), 101–130. https://doi.org/10.1037/0033-295X.115.1.101

Sawers, A., & Hahn, M. E. (2013). Gradual training reduces practice difficulty while preserving motor learning of a novel locomotor task. *Human Movement Science*, *32*(4), 605–617. https://doi.org/https://doi.org/10.1016/j.humov.2013.02.004

Shahar, N., & Meiran, N. (2015). Learning to control actions: Transfer effects following a procedural cognitive control computerized training. *PLOS ONE*, *10*, 1–22. https://doi.org/10.1371/journal.pone.0119992

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

Singley, M. K., & Anderson, J. R. (1987). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interacttion*, *3*(3), 223–274.

Squire, L. R. (1986). Mechanisms of memory. *Science*, *232*(4758), 1612–1619. https://doi.org/https://doi.org/10.1126/science.3086978

Squire, L. R. (2004). Memory systems of the brain: A brief history and current perspective. *Neurobiology of Learning and Memory*, *82*(3), 171–177. https://doi.org/https://doi.org/10.1016/j.nlm.2004.06.005.

Stearns, B., Assanie, M., & Laird, J. E. (2017). Applying primitive elements theory for procedural transfer in Soar. *International Conference on Cognitive Modeling*.

Stearns, B., & Laird, J. E. (2018). Modeling instruction fetch in procedural learning. *International Conference on Cognitive Modeling*.

Stearns, B., & Laird, J. E. (2020). Toward unifying cognitive architecture and neural task set theories. *Proceedings of the Forty-second Annual Conference of the Cognitive Science Society*.

Stewart, T. C., Choo, X., & Eliasmith, C. (2010). Dynamic behaviour of a spiking model of action selection in the basal ganglia. *International Conference on Cognitive Modeling*, 235–40.

Stewart, T. C., & Eliasmith, C. (2009). Spiking neurons and central executive control: The origin of the 50-millisecond cognitive cycle. *9th International Conference on Cognitive Modelling*, *122*(127), 130–131.

Sun, S., Councill, I., Fan, X., Ritter, F. E., & Yen, J. (2004). Comparing teamwork modeling in an empirical approach. *Sixth International Conference on Cognitive Modeling*, 388–389.

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

Taatgen, N. A. (2019). A spiking neural architecture that learns tasks. *International Conference on Cognitive Modeling*.

Thorndike, E. L. (1922). The effect of changed data upon reasoning. *Journal of Experimental Psychology*, *5*(1), 33.

West, R. L. (2020). Introduction to volume 1(2). *Common Model of Cognition Bulletin*, *1*(2). https://ojs.library.carleton.ca/index.php/cmcb/article/view/2703

Wharton, C., & Lewis, C. (1990). *Soar and the construction-integration model: Pressing a button in two cognitive architectures ; cu-cs-466-90* (tech. rep. CU-CS-466-90). Department of Computer Science, University of Colorado. Boulder, CO.

APPENDIX A

Soar Memory Systems

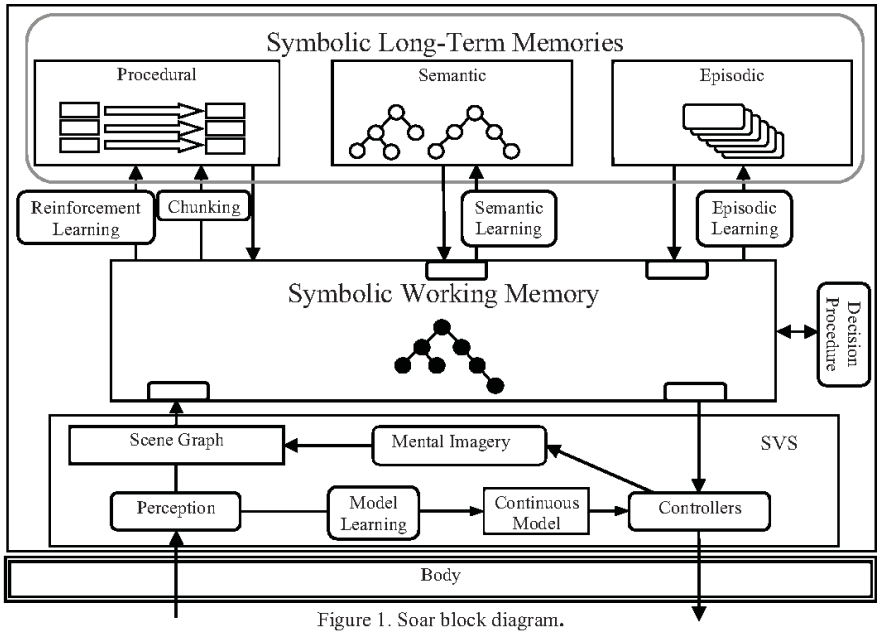


Figure A.1: The Soar cognitive architecture.

Soar is a cognitive architecture that defines many automatic mechanisms theorized to support cognition, as depicted in Figure A.1. Their theory and function is summarized as follows:

- **Working memory (WM):** A short-term store of information for active reasoning and problem solving, represented in Soar as a directed cyclic graph
- **Long-term procedural memory:** In Soar, procedural knowledge is represented with *if-then* production rules, which condition on WM and modify WM immediately in parallel whenever they match. New rules can be learned through *chunking*, an automatic process that summarizes practiced problem solving. Once learned, production rules are permanent in long-term memory and cannot be unlearned except by programmer intervention.

- **Long-term declarative memory (LTDM):** Explicit factual knowledge about the world, usually able to be verbalized (declarable) and manipulable by deliberate reasoning.
  - **Semantic memory (SMEM):** A store of discrete symbolic facts. In Soar, these are added and retrieved deliberately through decision making. These can be retrieved either by explicit reference or by query best-match. If by query, activation can be used to break ties between retrieval candidates that match the query equally well.
  - **Episodic memory (EPMEM):** A store of episodes experienced over time. In Soar, these are snapshots of WM stored automatically over time, but retrieved deliberately.
- **Decision making:** The process by which an operator is proposed, selected, and applied, defining a *decision cycle*. Procedural rules provide the proposals, selection preferences, and applications. The architecture performs the actual decision selection among proposed operators for each cycle, based on given preferences.
- **Subgoals** The theory of how WM interacts with decision making and procedural memory to organize executive function and goals. In Soar, this is represented through architecturally-created substate divisions in WM, rather than a separate component in the figure.
- **Perception:** The collection of raw stimuli from the environment and the processes that distill them into usable information provided to WM, implemented in Soar either with the Spatial Visual System (SVS) or direct input to WM. If perception comes through direct input, it is up to the modeler to ensure that environment input is in a form that is useful to the agent. Soar design focuses on cognitive processing, and generally assumes that perceptual processing is provided externally.

While Soar supports declarative knowledge representations and reasoning, it does not define an approach for how deliberate reasoning should take place, nor for how declarative knowledge should be acquired, structured, or manipulated in WM or LTDM. As a cognitive architecture, Soar is instead concerned with defining the automatic, fixed processes of cognition. It is up to a Soar programmer to design an "agent" that runs on top of the architecture and defines both the declarative reasoning process and any further automatic processes that structure the use of architectural mechanisms such as decision making or chunking. Soar's mechanisms for perception and LTDM in particular are only loosely driven by the architecture, and their function depends highly on agent design and commands written by the agent programmer within agent rules.

PROPs is built as a Soar agent with rules that both drive these architectural mechanisms and pass knowledge between them for task processing in a task-general way.

APPENDIX B

WM Theory and PRIMs

The original description of PRIMs theory defined WM to have a constrained small set of slots in order to support the existence of PRIM operations among them. Each unique PRIM rule in Actransfer is defined in terms of the WM slots that it operates upon. To make PRIMs theory work in Soar, I altered this aspect of the theory so that primitive rules did not depend on pre-defined knowledge of every possible WM element location, as I described in this chapter. In section 7.2, I defined a PRIM in the PROPs system as a primitive, innate, rule for a broad *type* of operation, which can apply to any WM content after PRIM resolution binds it to the desired content location.

One can argue that this changes the computational definition of a PRIM. If a PRIM is fundamentally defined as the most primitive level of rule, then PRIMs in Actransfer and PRIMs in PROPs are equivalent. But if a PRIM is defined as a primitive memory operation on specific memory elements, then a PRIM in the PROPs system is a very different thing. In the PROPs system, it is more accurate to say that the PROP, and not the single PRIM rule, is what actually operates on specific memory elements. (Though a PROP is of course defined using resolved PRIM rules.) The distinction between primitive rule and primitive memory operation on specific WM elements is undefined in original PRIMs theory, and thus I do not believe there can be a satisfactory answer as to which definition best matches the original intent at this time. I introduced the distinction out of computational necessity with Soar. Thus, my work represents a refinement of PRIMs theory. Using this new distinction, one can simply say that a primitive rule in Actransfer serves the role of both a PRIM and a PROP bound into one.

Is it reasonable to define the PROPs agent to support the full span of possible WM locations in Soar's unbounded WM graph? One can ask if perhaps WM has only a small number of slot addresses, perhaps four, the rough WM capacity limit suggested by Cowan (2001). Studies have shown that humans tend to have such a limit in how many distinct concepts we can hold in mind at once, and that while this can vary across individuals, it tends to be fairly constant for an adult across their lifetime (Cowan, 2010). Though the number four is close to the number of buffers in the ACT-R model, one might point out that it approximately corresponds to the number of slots typically used per buffer in Actransfer as well, and computationally, combinatorics would not be

a significant issue if WM was limited to a very small number of usable locations. However, it is important to remember that human WM capacity could be very different from the number of human WM addresses.

The general definition of human WM favored by Cowan (2017) is that it is the set of mental resources that are in a heightened state of availability for active processing, and Eriksson et al. (2015) point out that WM capacity is different from WM content representation. Human WM *content* is often considered to be the activated neurons and pathways that exist across the brain, particularly in the perceptual cortices, while WM *capacity* is thought to be an ability of the PFC and parietal cortex to maintain a specific activation pattern across those perceptual cortices (in spite of changing stimuli). The capacity for how many discrete, verbalizable concepts the brain can keep active at once in this manner seems to be a small number around four, but the number of possible WM content locations could be at least as vast as the full set of ways that the brain might represent perceptual input and its features and abstractions and then make these available for cognitive processing. Considering there are at least 200,000,000 cortical minicolumns in the human brain (Johansson & Lansner, 2007), the number of memory elements that could be activated must be vastly higher than three or four. While there is as yet no consensus regarding the exact role of the PFC and the surrounding regions in mediating WM activation (Eriksson et al., 2015), it is clear that WM in the human brain is not simply a copy of files moved from long-term storage to a short-term cache as in a classic digital computer. This means that WM in Actransfer represents something different than the human WM described by Cowan.

How would ACT-R and Soar definitions of WM map to this description of human WM? ACT-R and Actransfer limit the capacity of WM through a limited number of WM buffers and slots per buffer. In this model then, each WM element would seem to correspond with an *activation pattern* across the span of possible WM elements, and the limited number of slots represents the limited number of activation patterns that can be maintained at once. Soar, by contrast, models WM with an unbounded graph. Each graph element exists and is sustained in WM due to its connection with other elements. If any node in the Soar WM graph loses its connection to the main graph, then it is dropped from WM. In this model, the WM graph would seem to correspond with the broader network of actual memory elements that are activated across the human brain when they become activated for working use. The root of Soar's WM graph might then correspond with the PFC to the extent that it performs the role of maintaining content in WM.

What does this distinction imply for PRIMs theory? This would imply that an *ACT-R decision* (selected rule) represents an operation that modifies the top-level activation pattern in WM, whereas a *Soar decision* (selected operator) modifies the specific details of the network of activated WM elements. PRIM resolution follows from the deeper detail of the network, and corresponds with the work of propagating a decision for a top-level WM change down to the specific details of

the WM network.

Thus, these two paradigms might not be incompatible as models of human cognition, but rather they each focus on different parts of the same bigger computational picture. The ACT-R model of WM could even be simulated in Soar with a WM graph that is only ever one or two layers deep. And while Soar might support a deeper network of active WM elements, it is less constrained than ACT-R. ACT-R explicitly maps different WM buffers to different brain regions. Soar does not define whether or how different branches of its WM graph might correspond to different elements of the human brain. Indeed, at the time of this writing, there is no standard in Soar modeling that would promote any one particular structure of graph at all.

APPENDIX C

Implications of Procedure Contexts

Introducing procedure contexts to $PROP_3$ led to many subtle computational and theoretical changes in the model. I here walk through the implications of procedure contexts in greater detail than I did in the main body of the thesis, following the outline of P1-P6. For each phase, I describe how procedure contexts affect the processing and theory of that phase. I begin each section with a restatement of the challenge posed by that phase of PRIMs theory.

C.1 P1: Types of Procedure Context

What initial set of primitive procedures supports truly task-independent behavior?

The challenge of P1 is to define primitives for the agent in such a way that the agent is capable of task-independent procedural learning with human-like efficiency. The set of primitives must be task-independent for the scope of an agent's potential learning to also be task-independent. Further, the format of the primitives must define the rest of the agent processing, because it provides the fundamental structure for representing task knowledge in both declarative and procedural memory. The dilemma with Actransfer was that its implementation of P1 constrained the definition of WM to hold a fixed number of slots, which deviates from ACT-R and is also incompatible with Soar. In $PROP_3$, I use procedure contexts to define the computational primitives of procedural practice and learning, as I just illustrated with Figure 9.2. Procedure contexts can be used to represent the transcribe-text task as easily as any other task that could be performed with Soar, and are as task-independent as Actransfer instructions. But they also led to the following specific extensions to PRIMs theory related to P1:

1. I introduce a distinction for PRIMs theory between the representations of primitive conditions and actions, respectively.

2. I introduce a theoretical distinction that condition lines should be evaluated in parallel rather than through sequential decision making to support choice-based decision making. We go on to show that this can improve the scale of latency in human modeling.

I now describe these in detail.

C.1.1 Instructing Conditions vs Actions

In Actransfer's design, conditions lines and action lines trigger matching primitive condition and action rules, and in ACT-R theory a primitive rule corresponds with a primitive *decision*. This meant that the Actransfer agent used sequential decision making in order to evaluate each of the condition lines of each retrieved instruction. As I discussed when describing PROP_2, spending a decision for each condition line leads to significant latency overhead, especially when the agent retrieves an instruction with non-matching conditions.

The problem is that, in Soar, a primitive rule does *not* represent a primitive decision. Soar decisions are *operators*. Soar uses elaboration rules to propose operators and make relative preferences among them, and then uses apply rules to carry out selected operators. But PRIM condition and action lines must correspond to primitive rules, since they represent primitive memory operations for rule-based architectures. Therefore, PRIM condition and action lines in Soar cannot correspond directly to primitive decisions in Soar.

I introduce this distinction between the computational representation of condition lines and action lines for PROP_3, that condition lines instruct primitive elaboration rules and action lines instruct primitive apply rules. The agent then generates operators as decision choices on-line by using its PRIM elaboration and apply rules in its active state of WM.

PRIM condition and action lines thus represent different kinds of operations. Conditions as elaborations represent temporary changes to WM in a way that merely elaborates on other WM structures. Actions as applications perform lasting changes to WM in response to decisions. This leads to my distinction between elaboration contexts and apply contexts, which I described in the previous section. Each type of context instructs a different kind of memory operation for a different kind of WM state and goal.

Procedure contexts in PROP_3 are declarative graph structures, and each type of procedure context is a distinct kind of graph structure. Figure C.1 shows the "get-text" elaboration context and the "read-prompt" apply context from the transcribe-text task example. On the left in gray is the condition/action instruction logic for the "read-prompt" operator that is split across these two structures.

The "get-text" context `P2` includes instructions for two operator proposals, `L4` and `L5`. Each instructed proposal is defined in terms of the proposal *conditions* and the name of the operator to

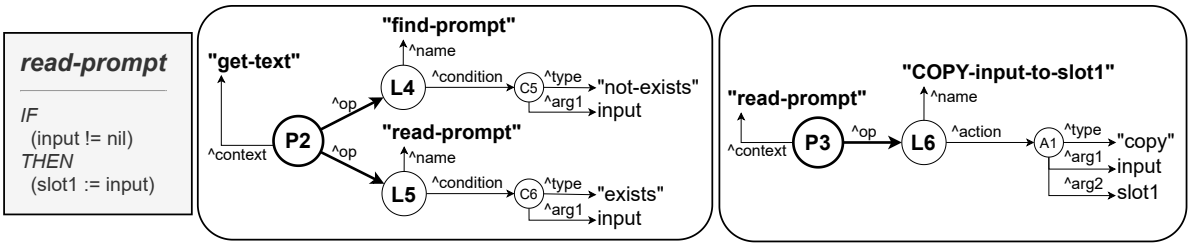


Figure C.1: The procedure context structures for "read-prompt" in the transcribe-text task.

be proposed. The "read-prompt" apply context, by contrast, instructs an operator application with `L6`. This instruction is defined in terms of an operator *action*. The "read-prompt" apply context includes no conditions, because these are instructed under "get-text." The agent will propose the operator named "COPY-input-to-slot1" as soon as it retrieves the "read-prompt" context into WM.

In this case, the "get-text" elaboration context includes two instructed proposals `L4` and `L5`, which each only have one condition, but in general any number of proposals could be instructed under `P2`, and any number of conditions could be included for any operator proposal. Similarly, `P3` in the "read-prompt" context points to only `L6`, but could point to any number of action lines.[1]

As I will describe in section C.1.2, the agent uses PRIM elaboration rules to evaluate all instructed condition lines in an elaboration context in parallel. If all conditions are satisfied for an instructed operator, a PRIM proposal rule proposes that operator as a choice for decision making. Once an operator is selected by decision making, the agent can then retrieve an apply context in a substate to get the action lines for the selected operator. In that substate, the agent uses further operators to apply each instructed action line. If the agent retrieves an apply context that instructs only a single action line, that action can be applied by a single PRIM apply rule, and the corresponding proposed operator is a primitive operator (PROP).

C.1.2 Evaluating Conditions in Parallel

One of the reasons classic production systems have long been desirable for modeling rapid choice-based human decision making is their ability to match production rule conditions in parallel. The production system has been very well established as a basis for modeling human reasoning ever since Newell introduced them to psychology (Newell & Simon, 1972). It is also a well established modeling approach with production system cognitive architectures to learn new procedural rules based on declarative task knowledge, following the declarative/procedural order of the three-phase theory of skill composition (Anderson, 1982). But more recently, as cognitive architecture models have grown to use increasingly general procedural rules, so also has the amount of condition/action

[1] In technical detail, `L6` in the figure represents an instruction to propose a primitive operator that will execute the attached action line. `L6` can only point to one action, but `P3` can point to any number of operator proposal instructions.

rule behavior that is instructed through declarative WM content grown, as demonstrated by PRIMs theory as well as less-general models in ACT-R and Soar Anderson et al. (2019) and Kirk and Laird (2019). The question of how one should properly process such declarative instruction knowledge is becoming increasingly important.

In PRIMs theory, if a PRIMs agent uses decision making to evaluate each condition line in sequence, it will be next to impossible for the agent to retrieve and test multiple competing instructions quickly enough to react to the kinds of rapid tasks humans face every day. Even if the agent retrieves the whole set of competing instructions together as I have $PROP_3$ do, even five conditions for a single instruction would require 0.25 sec to evaluate at 50 msec each with sequential decisions, not including the extra time that would be needed to first retrieve the set of instructions. The agent might be able to eventually learn to evaluate all condition lines for a single instruction together. This is what Actransfer does, so that eventually it only takes a single decision cycle to evaluate the conditions of any single retrieved instruction. But when the agent must compare multiple instructions and their condition lines together at once, even rule compilation cannot easily make decision making practical. And this approach still requires the Actransfer agent to have prohibitively slow performance when beginning a task that it has not yet learned. If the agent began a new task in which it had to choose from among four competing operations, each with only three conditions, the agent would potentially spend 0.6 sec over twelve decision cycles just to choose a single cognitive operation for the task. Humans by contrast show reaction times on the order of 0.25 sec for tasks once they understand their task instructions Jain et al., 2015.

I observe that this demonstrates a problem that can arise any time declarative WM content is used to instruct rule behavior for a production system. Production systems are able to evaluate rule conditions in parallel, but when an agent must use rules to interpret declarative condition lines in WM it might no longer be able to apply this parallel condition-matching ability toward task decision making.

My work developing $PROP_3$ therefore introduces a theoretical distinction between how cognitive models should process condition lines and action lines in WM, which should be applicable to any model of choice-based decision making that uses declarative instructions as a basis for rule-based decision making. For a cognitive model agent to be able to select among multiple valid rule behaviors with a speed classically expected from a production system's parallel rule-matching capabilities, the agent must be able to evaluate the instruction conditions with the same speed that a production system can match rule conditions. Therefore, the agent should not be constrained by the decision cycle bottleneck in order This distinction is a minor contribution of this thesis.

In Soar, elaboration rules that propose and give preference to operator choices fire in parallel and do not require their own decision cycles. Thus, in $PROP_3$ PRIM elaboration rules evaluate all condition lines continuously in parallel whenever they are in WM in an elaboration context.

This makes rapid task responses tenable even when the agent is first exposed to a task and has not yet practiced it. In $PROP_3$, at any time if any one condition line for any operation included in the elaboration context becomes newly satisfied or no longer satisfied, the PRIM elaboration rules that evaluate those conditions immediately fire or retract accordingly to mark a condition line as satisfied or not. Action lines must still be applied through serial decisions and apply rules, not through parallel elaborations, as I will discuss in section C.4. The simple reason in terms of Soar theory is that these must make long-term changes to WM.

Though consistent with the overall PRIMs processing flow, evaluating conditions in parallel is a modification from the published description of PRIMs. However, recent independent work by Taatgen (2019) that attempted a neural basal ganglia model of PRIMs implied a similar distinction between condition and action lines, namely that conditions had to be processed with different mechanisms from action lines and in parallel. All together, this distinction between condition and action lines might imply that elaboration contexts and apply contexts correspond with different processing regions in the human brain. Elaboration contexts might correspond better with task sets in the PFC while apply contexts more with pre-motor, motor, or basal ganglia representations. Future work should investigate this question further.

Recall from section 4.2 that PRIMs conditions and actions are task-independent because, like computer assembly operations, they are defined based on the WM locations that they use, not the task-specific contents in those locations. Recall that PRIMs theory therefore separates condition and action lines from the task-specific values that can be loaded into them, for example by treating a condition such as (`IF input == "eol"`) as requiring two separate primitive operations, first a primitive that loads the task-specific value `"eol"` into a reserved general slot[2](call it `const1`), and then a task-independent condition primitive that tests (`IF input == const1`). This latter condition primitive is then able to transfer across any instructions that reference the `const1` slot, regardless of the contents of that slot. In the same way that Soar's distinction between elaboration and apply rules lets $PROP_3$ evaluate conditions in parallel, it also lets the agent retrieve task-specific, or rather, context-specific values in parallel along with the elaboration context content, as described in section 9.1.1.

C.2 P2: Retrieving Procedure Contexts

How does the agent select a specific instruction when retrieving from LTDM?

I describe the following four contributions of my work with $PROP_3$ and procedure contexts as

[2]Refer back to the red PRIM in Figure 4.4 on page 30, which illustrates the primitive that loads task-specific values in the original formulation of PRIMs theory.

related to P2:

1. I show how task context can remove the need for the PRIMs agent to search through its LTDM for a relevant task instruction.

2. I introduce a theoretical distinction between when spreading activation is or is not appropriate as a basis for making choices available to executive decision making.

3. I identify cognitive phase learning with PRIMs in Soar as the process of arranging a declarative hierarchy of task instructions in SMEM.

4. I propose the more general theory that cognitive phase learning is the process of arranging a task set hierarchy of condition/action instructions.

C.2.1 Searchless Retrievals

I discussed the P2 retrieval selection problem in detail in the previous chapter about $PROP_2$. As stated earlier, I saw two approaches to implementing the P2 process in Soar. According to step 3.b.2 of my methodology, I implemented and evaluated each. The first spreading activation approach I tested with $PROP_2$, and the second Soar decision making approach I applied with $PROP_3$.

The $PROP_2$ agent selected satisfied instructions via spreading activation, and used a network of memory associations that was independent from decision making to determine instruction activation. In other words, as far as executive control was concerned, there was no explicit WM knowledge about or executive control over what instructions the agent might retrieve and practice next. However, using procedure contexts in $PROP_3$, the agent uses decision making to deliberately follow the links among its procedure contexts and access instructions for different task operations. This eliminates the need for blind searches through LTDM.

I already described the hierarchical retrieval and decision making process in $PROP_3$ in section 9.1.1 with the example of Figure 9.2. Assuming an overall task is first given to an agent, a procedure context specific to that task can be explicitly retrieved for the starting goal. This must be an elaboration context, since instructed rule behaviors must begin with condition lines. If the agent is in a domain that involves task-switching, if the agent is given knowledge ahead of time that there are multiple tasks it will need to switch among, then the starting elaboration context can itself point to other elaboration contexts specific to each of the different tasks. Each task can thus be applied as its own operator within the main task-switching task.

C.2.2 Distinguishing Types of Retrieval Selections

In a real-world environment, there must be times when the agent's active procedure context is not sufficient to let it retrieve the instruction it needs. For example, what if the $PROP_3$ agent was working on the editors task but suddenly the computer screen started showing it the arithmetic task prompts? The tree of editors task procedure contexts does not link to the arithmetic task procedure contexts, and the agent would not be able to deliberately retrieve the instruction that it needed in order to respond. Or in a more extreme case, what if the agent was working on the editors task and suddenly a bear walked into the room? How would the agent switch from processing the expected editors task stimuli to processing the unexpected stimuli of the bear? Decision-based access to instructions in LTDM might let the agent perform more quickly and avoid distractor instructions, but this cannot work when the agent has no connections from WM to the instructions it needs. Clearly there must be a difference between how PRIMs instructions can be accessed for expected and unexpected stimuli.

I introduce this distinction to PRIMs theory for P2 between accessing instructions for *expected* and *unexpected* stimuli within an agent's context. In this case, "expected" means that the $PROP_3$ agent has WM knowledge that would let it directly retrieve a procedure context in response to stimuli or a task prompt. For expected stimuli, the agent has instructions in WM that have condition lines that match the given stimuli and let the agent pursue connections through LTDM to various possible responses and strategies. But for *unexpected* stimuli, the instructions that the agent has in WM are irrelevant to the current stimuli, and do not provide meaningful operator proposals. The agent cannot use decision making to select a specific instruction from LTDM and must rely on a different, non-deliberate process for P2.

Whenever the agent is presented with unexpected stimuli, and must therefore switch procedure contexts without the benefits of following direct links in WM, this is when it would make sense, and even be necessary within this computational formulation, for the agent to use architecture-driven, activation-based retrievals. For each case, a different type of P2 process is computationally more advantageous. For expected stimuli, the learner can perform its tasks more quickly by deliberately following links in WM. For unexpected stimuli, the agent must rely on the architecture's ability to use LTDM associations from the stimuli to activate relevant instructions.

The distinction between expected and unexpected stimuli leads to a distinction between *deliberate* and *non-deliberate* P2 processing. This distinction is specifically in whether the P2 process of *selecting* an instruction from LTDM depends on the agent's deliberate decision making or not. This does not assert whether the agent initiates a retrieval deliberately or spontaneously. In the case where P2 is non-deliberate, the agent could hypothetically have used deliberate decision making to initiate P2 to retrieve *something* based on activation. But when P2 is deliberate, the agent knows exactly which instruction bundle it is trying to retrieve into WM and can thus avoid the latencies

or uncertainties that can come with a spreading-activation approach.

Given this distinction, a process for responding to unexpected stimuli would clearly not be relevant within a model of learning to perform procedures through practice, where procedures are instructed before the learner begins its tasks. My thesis is specifically concerned with the process of procedural learning and practice given that the agent already has task instructions that it can practice and learn. The above distinction shows that non-deliberate retrievals can be necessary for *beginning* a task's procedures and accessing the starting WM context for the task, but they are not necessary for *learning to execute* those task procedures, and thus are, in fact, outside the scope of my work.

The integration of deliberate P2 using procedure contexts with non-deliberate P2 using spreading-activation presents an intriguing avenue for future work. Non-deliberate procedure context retrievals might conceivably be used to model the phenomena of surprise (when non-deliberately retrieved procedure contexts have nothing in common with existing procedure contexts in WM), creativity (when non-deliberately retrieved contexts unify with existing WM contexts in unexpected ways), or even mind-wandering (when non-deliberately retrieved contexts interfere with using the existing procedure contexts in WM).

If one were to implement non-deliberate P2 in PROP_3, one potentially straightforward solution would be to treat a Soar *state no-change* impasse as a prompt for a spreading-activation based procedure context retrieval. A state no-change impasse occurs in Soar whenever the agent has no operator proposals for its task state. This is precisely what would occur if the agent's active procedure context was irrelevant to the current stimuli. Then the agent would need only associations between the stimuli and the elaboration contexts that conditioned on that stimuli, as I demonstrated with PROP_2. One difference with the PROP_2 model in this case, however, is that the agent would only need to select an elaboration context from among those that *start* a task. For instance, if the agent was faced with prompts for the transcribe-text task, it would only need to consider associations between stimuli and the starting "Transcribe Text" elaboration context, not the "get-text" or "write-text" elaboration contexts.

I believe the problem of unexpected stimuli is a manifestation of the goal re-activation problem (Li, 2016), which is the problem of how an agent can remember to re-activate particular goal driven behavior that it has suspended, cleared from its WM, and stored in its long-term memory for later. Li, 2016 shows that spreading activation for spontaneous retrievals is a computationally superior approach to this problem compared to preemptive retrieval strategies. The addition of a spontaneous retrieval mechanism into Soar is another possible solution to solving P2 for unexpected stimuli.

C.2.3 Defining Cognitive Phase Learning in Soar

In the previous chapter, I discussed how $PROP_2$ learned P2 instruction retrieval selection automatically through practice, and how this corresponds well with the cognitive phase of three-phase theory. It makes sense to model cognitive phase learning with a non-deliberate learning mechanism when only dealing with unexpected stimuli, since this requires a non-deliberate P2. However, when using deliberate P2 to deal with expected task stimuli, as I do with $PROP_3$, cognitive phase learning must represent something different. For the $PROP_3$ agent to learn how to deliberately access its procedure contexts, it must learn when to follow links between procedure contexts as well as where those links should point.

Thus, for $PROP_3$, cognitive phase learning of P2 would be a process in which the agent composes or modifies the hierarchical instructions that it keeps in LTDM. This means learning the correct condition lines for each operator as well as which known action lines to connect them with. This must be a declarative learning and reasoning process.

More generally in Soar terms, one can assume that cognitive phase learning corresponds with learning when to propose and select operators within particular problem spaces. This means learning both the proposal conditions and which particular operator actions go with a proposal. Associative phase learning is then learning how to apply each operator. Autonomous phase learning is learning to fire both proposal and apply rules to perform a task without the aid of declarative task instructions in WM.

Assuming the $PROP_3$ model is a decent model of human task sets, we can then even more generally propose a unification between three-phase and task set theories, that the cognitive phase involves the process of arranging task set structures for a task.

In the $PROP_3$ framework, this would also imply that different subjects would form different procedure context instruction hierarchies for the same task. This ought to be a significant if not primary factor for defining individual differences in the PROPs system. Some learners might create more efficient procedure context hierarchies than others, in which they can access task responses to various stimuli with only one or two retrievals, while other learners might create more distributed hierarchies of procedure contexts that require more retrievals to navigate, but which might also present fewer distracting choices in WM at a time.

C.3 P3: Decision Choices from Procedure Contexts

How does the agent evaluate and choose to apply a retrieved instruction with decision making?

There are two primary contributions of using procedure contexts with respect to P3. The first

fills the P2/P3 implementation gap by allowing the agent to employ decision making over choices. The second incorporates Soar's RL to allow task-independent decision making. These are already described in chapter 9, but I discuss them here in greater detail.

C.3.1 Hierarchical Instructed Decision Making

Recall how Figure 9.2 depicted three procedure contexts together in WM across for three hierarchical goals in the transcribe-text task, "Transcribe Text," "get-text," and "read-prompt." Figure C.2 shows this same arrangement with greater detail with respect to the Soar PSCM, mirroring Figure 6.3 on page 55. For each subgoal, Soar creates a distinct WM state as a partition of the total WM graph, and the agent loads a single procedure context into each, and each procedure context describes operators for the specific problem space associated with that subgoal. In the first WM state, on the left in the figure, the procedure context ("Transcribe Text") describes three operators, each of which have their own conditions. In the example, assume that only "get-text" and "finish" have satisfied conditions, so the agent only proposes those two operators. Of these, the agent selects "get-text". The rest of the figure is as in Figure 6.3. The agent selects the "read-prompt" operator in the next substate, and applies this with the "COPY-input-to-slot1" operator in the final substate.

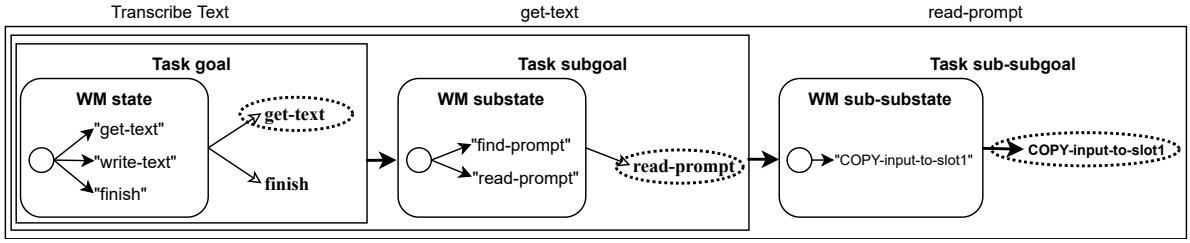


Figure C.2: Procedure contexts as PRIMs instructions in WM, according to the Soar PSCM.

The agent uses elaboration contexts such as "Transcribe Text" and "get-text" to structure its task decision making and condition-based reasoning. Once the agent retrieves an apply context, such as "COPY-input-to-slot1" as shown, the agent has chosen a specific task action. Apply contexts include no condition lines; the existence of an apply context in WM marks that the agent has already tested any relevant conditions and has chosen to execute the retrieved actions. From that point on until the instructed actions are complete, the agent does not use decision cycles to *choose* task actions but to *execute* them. The only reason the agent would stop performing the actions is if the task environment changed unexpectedly to interrupt the agent from its goals before it finished.[3]

[3]This is a feature of Soar. If the task conditions change so that the subgoal becomes irrelevant, Soar will immediately remove the subgoal and any contents of the associated substate, and the agent will resume decision making in a higher-level subgoal to respond to the task change.

Given that the agent has multiple instructed operators in an elaboration context that all have satisfied conditions, and the agent proposes each of them, how does the agent know which proposed operator to select? Recall that Soar normally uses *preference rules* to determine which proposed operators to select. Also recall that there are two main kinds of preference in Soar, symbolic logical preferences (If `op1` and `op2` are both proposed, prefer: `op1 > op2`) and utility-based preferences that can be learned via RL (`op1-utility = 0.54, op2-utility = 0.37`). $PROP_3$ supports both. Logical preferences can be instructed in elaboration contexts just as any other elaboration rule. There is little more to say regarding this kind of preference support, because this functionality comes for free with Soar without much extra effort needed in the $PROP_3$ agent design. The second utility-based preference-learning approach, however, requires a bit more explanation.

C.3.2 Considerations for PRIMs-based RL

In section 9.1.3 I described how I combined PRIMs theory with Soar for task-independent RL value function features. I mentioned that the use of condition lines as value function features does not inherently support the ability for the agent to be influenced by features that are not part of a proposed operator's conditions. For an environment feature to influence $PROP_3$ decision making, that feature must be included in the condition lines of the relevant operators in some form.

For example, assume an agent is in a maze task, and has an elaboration context that instructs three operators when the agent is at a junction in the maze, one each for traveling either left, straight, or right. The condition for each proposal is that that particular direction is not blocked. But if the agent reaches a junction, call it junction X, and while there it can see that at the end of the left path there is a trove of treasure! Since the condition lines for each proposal only test whether the path is blocked or not, the agent would not be able to add utility based on the treasure trove. In theory, in $PROP_3$ one could give an agent designer the ability to program RL-specific condition lines into an elaboration context, such that these would only affect operator utility and not whether an operator was proposed. It is uncertain whether this is necessary or even desirable for modeling human cognition, however. The above limitation results only from how the agent reasoning is programmed. For instance, in the maze example, one does not have to represent the elaboration context as I described above, with three proposals for the three directions and a single condition for each. A more deliberate representation of maze reasoning might be to have an elaboration context that instructs three operators for *considering* either the left, straight, or right paths if those paths are not blocked, with each of these then leading to a substate in which the agent looks down each path to see if one is better than another. In this case, the decision making is not determined by non-deliberate utility or RL, but by deliberate reasoning, and this is arguably what most humans

would do when traveling an unfamiliar maze.

RL for learning utility is for learning *automatic preference* without the need for deliberate investigation. What might this look like in the maze example for a human model? If the maze agent practiced a particular maze often enough to know that the left path at junction X always led to treasure, then a human-like agent might learn to travel left automatically at junction X without the need to investigate each path first. Notice that this scenario changes the features that the agent considers while traveling the maze. The agent no longer chooses the left path because it sees treasure there but because it sees and remembers the specific junction. In a PROP_3 agent, this implies that the agent would learn a modified elaboration context that had the agent test the junction rather than the sight of each path. For example, the agent might learn a new proposal that it added to the three mentioned above, along the lines of "IF (at junction X) THEN (propose routine go-left)," and the agent could learn a high utility for this operation after doing it successfully many times. The fact that the agent reacts to different environment features after learning in this example strongly implies that a significant part of this kind of learning is not merely from tuning utility but also the rearranging the declarative understanding of the task. Thus, I hypothesize that declarative learning needs to work in tandem with RL to create or modify proposals so that they use relevant condition lines. For a further example, consider a human in the maze who had practiced it often enough to have an automatic routine for charging around the left corner at junction X without looking first. Assume that one time the human then found he or she charged right into a monster that was right around the corner guarding the treasure. That person would likely not make the same mistake next time. They would immediately change their routine so that they would consider looking around the corner carefully before choosing it, even if expecting treasure to the left, because they also expect the possibility of a monster. This kind of one-shot learning implies the need for declarative learning to work in tandem with any RL system in order to make the proposal instructions consistent with the relevant task features.

C.4 P4: Hierarchical Apply Contexts

How does the agent use known procedures to execute a selected instruction?

P4 is the PRIMs theory phase in which the agent uses known rules to execute the instruction selected in P3. This was already well-defined in Actransfer, using ACT-R's mechanic for decision making and rule execution. This was also already defined in PROP_1 and PROP_2, using Soar's mechanic for operator selection and application. However, the PROP_3 procedure context structure extends the theory and computational detail of P4 in my model in two ways:

1. I integrate recursion into PRIMs theory to naturally support hierarchical task reasoning.

2. I refine the PRIMs theory distinction between task-independent instruction lines and task-specific values to specifically a distinction between task-independent action lines and task-specific action values.

C.4.1 Recursive PRIMs Processing

Because Soar problem spaces are hierarchically nested, procedure contexts introduce the notion of recursion into PRIMs processing. Specifically, a $PROP_3$ agent can implement P4 within the PRIMs processing pipeline by invoking another PRIMs processing pipeline. Whenever the agent selects an operator and the agent does not know a rule/chunk that can immediately apply it, the agent enters a new Soar substate and restarts the PRIMs processing cycle by retrieving another procedure context. When the agent does already know an apply rule(s) or chunk(s) that can immediately apply the selected operator, and thus carry out P4, this is the terminating condition for the recursion. The agent can use the apply rule directly without the need for further declarative instruction.

C.4.2 Value Contexts

In section C.1.2 I described how I include `consts` structures within elaboration contexts to hold context-specific values for condition lines, such that the $PROP_3$ agent does not need to perform additional retrievals to access these values. This does not work for apply contexts.

The basic reason is that the apply rules that the agent learns must remain task-independent and distinct from the context-specific values that they employ. Recall that chunking summarizes the processing of particular substates, and in $PROP_3$, specifically summarizes the processing instructed by apply contexts. If the agent included context-specific values in the same substate as the apply context action lines, then those context-specific values would be included in the chunked combinations of those action lines. Thus, the chunks of action lines would no longer be task-independent. So context-specific values *cannot* be in the same procedure context, or the same substate, as the task-independent action lines.

Actransfer's implementation retrieves task-specific contexts separately from condition or action lines for similar reasons. Actransfer has to use a separate PRIM to access task-specific values so that they do not become embedded within the learned rules that combine general condition and action PRIMs.

There are two apparent ways that I could have the $PROP_3$ agent access these values. The first is that I include them in the elaboration context that immediately precedes an apply context, in the same `consts` structure that is used for the condition lines. There are two downsides to this approach: Foremost is that, as I will describe further in section C.5, this would prevent the agent from being able to learn chunks that have context-specific values embedded within them, which is

prescribed by PRIMs theory in Taatgen, 2013. Additionally, most of the time these values will be irrelevant to what the agent is processing in the elaboration context's substate. Computationally this means they would unnecessarily take up memory and processing resources. The second approach is to observe that the context-specific values that are needed for apply rules represent their own context in between the conditions and the general application memory operations, and to therefore treat them as such. I believe this latter option is more theoretically sound, while also more efficient with memory.

I therefore differentiate two types of apply context: action contexts and value contexts. **Action contexts** encode the task-independent action lines for an instructed task operation. **Value contexts** encode the context-specific values for a group of action lines. Action lines in $PROP_3$ are always defined using a *single value context* as a header in a substate that precedes *one or more hierarchically-arranged actions contexts* and their corresponding substates. A value context only ever points to a single named action context that defines the top of the general action line hierarchy.[4]

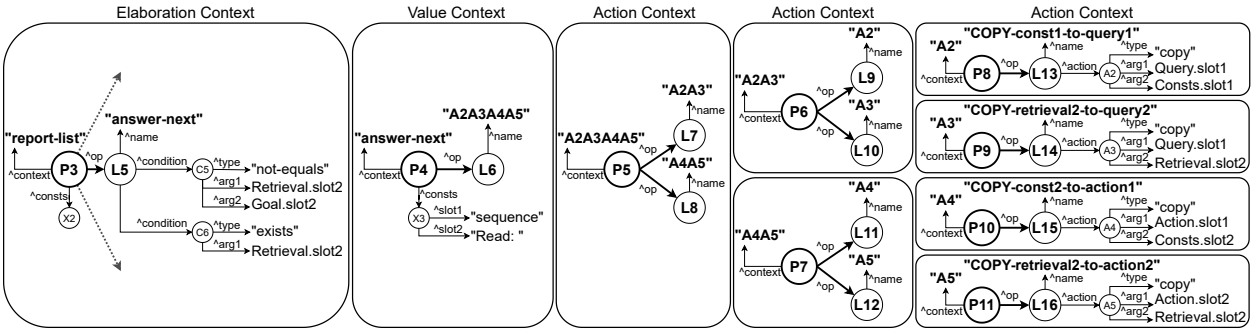


Figure C.3: Procedure contexts for the "answer-next" instruction depicted in Figure 4.4.

Figure C.3 shows the hierarchical procedure context representation of the "answer-next" instruction from Figure 4.4 on page 30. The two conditions are in the elaboration context on the far left, the context-specific constants for thee action lines are in the value context to its right, and the four COPY operations are in the hierarchy of action contexts to the right of that. In the figure, each layer of procedure contexts from left to right represents a deeper layer of Soar substate.

The figure above shows the four action lines arranged in a specific binary hierarchy across three layers of substates. This hierarchy of substates and subgoals leads to the binary hierarchy of learned rules as prescribed by PRIMs theory, as discussed in section 4 and depicted in Figure 4.4, which allows general transfer of compositions. The $PROP_3$ agent learns chunks for pairs of action lines that are clustered together in the same action context.

[4]For simplicity, previous examples such as Figure C.1 show a "read-prompt" elaboration context leading directly to a "COPY-input-to-slot1" apply context, without an intermediate value context. A real $PROP_3$ agent would always require a value context.

Apply contexts do not include their own condition lines. When the agent retrieves the "answer-next" value context, it immediately proposes the "A2A3A4A5" operator, and then immediately selects it. When the agent then retrieves the "A2A3A4A5" context, it immediately proposes both of its described operators, "A2A3" and "A4A5." $PROP_3$ gives these two operators *indifferent* preference in Soar, which means that the architecture will choose one at random to pursue first. Order does not matter within action contexts. $PROP_3$ will buffer their effects to be carried out in parallel after the agent finishes interpreting all action lines.

In Figure C.3, if the agent selected "A4A5" after "A2A3A4A5," it would then retrieve the "A4A5" action context in another substate, and would then propose "A4" and "A5." If it then selected "A4" and retrieved that primitive action context, the PRIM apply rule for the COPY operation would then match and fire using the arguments given in the "A4" context using the values of the "answer-next" value context. The agent would then proceed to "A5." After similarly resolving the "A5" PRIM, the agent would then proceed to "A2A3," and so on in depth-first progression through the action contexts.

C.5 P5: Gradually Learning Apply Contexts

How does the agent compose practiced procedures into new procedures?

The same procedure context structures in $PROP_3$ that define declarative task instructions and decision making also define the course of Soar chunking for gradual procedural learning. This is because Soar chunking is based on summarizing problem space computation, and procedure contexts define problem space computation. This resolves a significant problem that burdened the implementations for $PROP_1$ and $PROP_2$, which was that those agents had to use deliberate agent reasoning to trigger and control the chunking process.

I do not introduce any new changes to Soar for chunking in $PROP_3$. I use the same gradual learning extension that I introduced to the architecture with $PROP_2$. I show how the procedure context processing that I described earlier in this chapter leads to the following additions to the overall model and theory:

1. I show how PRIMs theory can integrate naturally with Soar's problem-space computational model to allow automatic hierarchical chunking of PRIMs instruction lines.

2. I distinguish that a model of PRIMs theory for procedural learning should not inherently include cognitive phase learning

3. I distinguish that a model of PRIMs theory for procedural learning should generally omit the autonomous phase learning.

4. I identify shortcomings in both ACT-R and Soar cognitive architecture theory that currently prevent either from fully supporting the model of gradual, hierarchical procedural learning and execution implied by this research.

C.5.1 Unifying PRIMs and Chunking

As described in section 6, when decisions in a subgoal lead to changes to a parent goal's WM state, Soar's chunking mechanism learns new rules, called chunks, that summarize those changes and repeat them automatically in the future. The problem with implementing P5 for $PROP_1$ and $PROP_2$ was that PRIMs learning was based on ACT-R's sequential production compilation model, where the architecture compiles procedures that are practiced back-to-back, and was not directly compatible with Soar chunking.

ACT-R's production compilation could be described as a type of bottom-up learning. In that model, the agent learns to cluster primitive operations together *after* repeated practice. Soar's chunking, on the other hand, requires a certain degree of top-down structure. A Soar agent learns rules based on the structure of impasses and goals that *precede* substate operator execution. A Soar agent *cannot* create a subgoal, and thereby learn a new chunk, unless it first identifies a specific impasse that gives the learning process, and any learned chunk, its structure.

Soar impasses and subgoals are supposed to be architecturally-managed processes that respond to the nature of the task problems and goals, and should not generally be based on the agent deliberately controlling its own architectural learning. A proper model of human procedural learning should be a model of tacit learning. But in order for $PROP_1$ or $PROP_2$ to learn chunks for pairs of primitives based on practice, I had to have the agent use decision making to collect pairs of practiced operators and deliberately create impasses and subgoals so that it could compile these together through chunking.

In $PROP_3$, the agent does not need to deliberately coerce its own impasses or chunking. The hierarchical structure of procedure contexts in SMEM provides the structure that the Soar agent needs to automatically respond to impasses and learn chunks for PRIMs processing. The $PROP_3$ agent's decision making focuses only on the problem space of the *task*, not the problem of creating impasses. The Soar architecture then automatically recognizes the task impasses when the agent lacks rules for proceeding in the task, creates subgoals for solving those impasses via declarative instructions, and learns chunks for performing those task instructions.

For example, consider Figure C.4, which shows the chunks that the $PROP_3$ agent would learn after repeatedly practicing the apply contexts shown back in Figure C.3. The $PROP_3$ agent will learn chunks that summarize the effects of each of the depicted apply contexts, from right to left with subsequent iterations of practice. At the lowest level, the $PROP_3$ agent chunks primitive

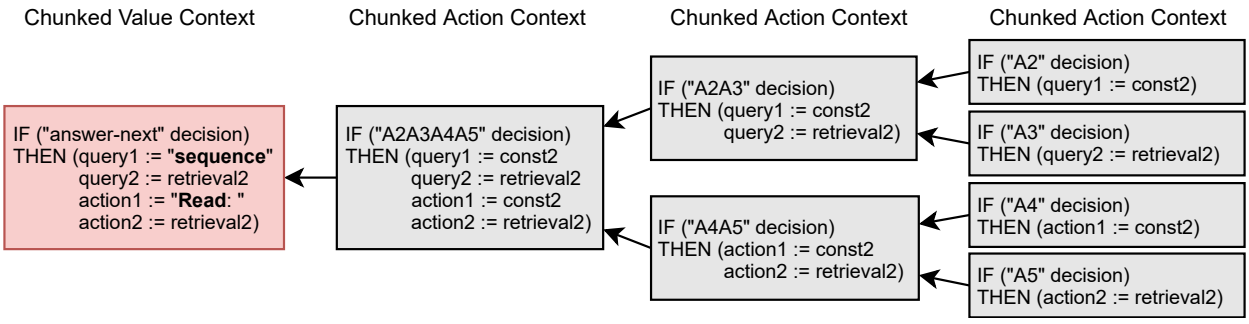


Figure C.4: Chunks learned for the "answer-next" apply contexts depicted in Figure C.3.

action context operations, such as "A2" through "A5" in the figure. This is when the agent learns automatic PRIM resolution for the primitive actions that the agent actually uses in practice. These chunks are the most transferable, because they can be applied for any instructions where the same action line is used. Next, the agent chunks groups of actions that are paired together in higher-level action contexts, such as "A2A3" and "A4A5" in the figure. These are transferable wherever the same pair of action lines is used together. After learning a chunk for "A2A3" under the answer-next "A2A3A4A5" action context, the agent might also use the "A2A3" chunk under another context such as "A2A3A8A9." Eventually the agent compiles a chunk that can apply the highest-level task-independent action context in the hierarchy, which in Figure C.3 is the "A2A3A4A5" action context. This chunk can transfer to apply any task instructions that use the same general memory operations but any context-specific values. The final level of chunking a PROP_3 agent can do is, as prescribed by PRIMs theory, to compile the context-specific values into the task-independent rules that apply the instruction. This happens when the agent chunks the level of value contexts. The result of learning a value context chunk is that the agent is then able to apply the operator directly from elaboration context state, without retrieving value contexts or action contexts. In this case, after the agent learns the "answer-next" value context, it will then be able to instantly apply the "answer-next" operator whenever that operator is proposed and selected in the future.

Actransfer as described by Taatgen (2013) compiles values after compiling both condition and action lines together. If an Actransfer agent encountered a task instruction that used the same action lines as another instruction it had already learned, with the same task-specific values, but the instruction used different condition lines, that agent would not be able to transfer the prior learning of compiling the same task-specific values together with those actions. It would have to first compile the task-independent conditions and actions together, and then embed the values into that compilation.[5] However, because PROP_3 separates condition and action lines into separate

[5]Compiling values into rules was disabled in a later revision of Actransfer. In that model, the agent would use this final layer of learned rules in place of instruction, and this would often then prevent decision making flexibility. (N. Taatgen, personal communication, July 11, 2017)

problem spaces, the chunk that applies "answer-next" in the above example could be used to apply that operator no matter what the condition lines were that led the agent to propose it. The $PROP_3$ agent could theoretically have any number of elaboration contexts that propose the "answer-next" operator, each for different reasons, but the same chunk could apply the operator in all cases. This is a level of transfer that Actransfer does not support.

The $PROP_3$ agent cannot learn chunks that would summarize elaboration contexts or otherwise prevent the need for it to retrieve them. Elaboration contexts give structure to task processing that is temporally-extended, not parallel. Recall, for instance, the transcribe-text task example. In that task, the agent must first read a text prompt before transcribing that text into a computer text editor, and no amount of practice can change that sequential dependency. It would not make sense for the agent to chunk the task knowledge of elaboration contexts any more than it would for an agent to perform the entire transcribe-text task from start to finish with only a single cognitive cycle – the task requires temporally-sequenced interactions with the environment. And, as I already described in section C.1.2, the nature of elaboration rules in Soar means that the $PROP_3$ agent can process condition lines from elaboration contexts instantly in parallel anyway, such that there is no practical benefit from chunking them.

C.5.2 A Shortcoming: Learning Transfer Structure

Notice that one substantial difference between $PROP_3$ and prior versions of PROPs is that $PROP_3$ does not learn the composition structure of the PRIMs hierarchy during practice. The hierarchy of apply contexts is provided in SMEM before the agent practices any action lines. In $PROP_1$, and $PROP_2$, this hierarchy was learned implicitly during practice. $PROP_3$ makes this hierarchy explicit before practice. This is computationally necessary for $PROP_3$ because Soar substates need the structure of goals and impasses before any substates can be created, and therefore this is necessary to implement P2 and P3 for PRIMs in a manner consistent with the Soar PSCM.

This means that with $PROP_3$ I am constrained to use a similar pre-processing approach as Actransfer to make task instruction lines more transferable. That is, when I generate the declarative instructions to put into the agent's SMEM before running it on any task, I have a program scan all sets of action lines and determine ahead of time which binary clusters appear most often across all instructions. This lets the program generate instruction hierarchies that maximize transfer of action line clusters. This is essentially the same pre-processing done by Actransfer, with the main difference that Actransfer pre-arranged condition lines this way in addition to action lines.

As I described in section 7.5, a computational motive for gradual procedural learning in the first place is so that the agent can learn how to hierarchically arrange instruction lines based on *experience*. If this hierarchy is pre-arranged as done explicitly in $PROP_3$ or implicitly in Actransfer,

then there is little apparent motive for gradual learning other than to generate the appearance of human-like behavior. But the computational principles of ACT-R or Soar seem to require this kind of pre-processing step, if requiring rigorous consistency with the overall theory of the architectures.

What would it take to have an Actransfer agent learn the instruction line hierarchy while also being consistent with the surrounding architecture? In what ways would ACT-R theory have to change? The ACT-R architecture's production compilation mechanism provides a method that could learn pairwise combinations of rules based on experience. This mechanism is what defined the learning approach of $PROP_1$ and $PROP_2$. However, to actually use this in ACT-R or Actransfer, the agent has to actually practice a pair of instruction lines together in order for the architectural learning mechanism to register that pair as *experienced.* Because ACT-R equates each decision with a single fired rule, this is impossible for ACT-R. The agent has to practice only a single ordered sequence of instruction lines, and the architecture will only experience pairs that are found in that sequence. It would take an impractical amount of practice for the agent to try all permutations of ordered instruction lines so that the architecture could compile the most transferable pairs. Further, the learning rate would almost certainly have to be set to be very slow to allow this extended time of practice. Thus, in order for an Actransfer agent to learn the instruction line hierarchies in a practical, gradual, on-line manner, the architecture would need to be able to register pairs of instruction lines as "experienced" and eligible for combination even when the pair was not practiced back-to-back through sequenced decisions.

How is Soar different? Soar allows instruction lines to be treated as sets rather than sequences. Plus, Soar can fire many rules per decision cycle. $PROP_1$ and $PROP_2$ were able to take advantage of this and have the agent fire many rules in parallel for each pair of instruction lines that occurred together during a single round of practice, and thereby register each pair as "experienced" and eligible for compilation. The problem was that Soar did not register this experience based directly on the rules firing together. Soar chunking compiles the net processing of substates. In $PROP_1$ and $PROP_2$ I had the agent create different declarative structures from a single substate, each of which corresponded to a different pair of practiced instruction lines, such that the architecture would recognize each created declarative structure as a different substate result and therefore learn a different chunk for each. This is a convoluted hack of a way to use the Soar architecture, and I do not ask the reader to fully grasp that approach here. I mention it here to emphasize the point that I was not actually having the Soar agent learn chunks for what it practiced for the task. The agent rather learned chunks for declarative meta-structures about its own learning, separate from its task practice. If I wanted Soar chunking to simply replicate the actions of the rules that performed the task processing, then Soar would summarize those actions into a single rule, not a hierarchy of pairs of rules. At the end of the day, just like ACT-R, Soar would only register the experience of rules that were actually practiced together in a substate, not all the different ways of pairing them

together.

What would it take to have a $PROP_3$ agent learn the instruction line hierarchy while also being consistent with the surrounding architecture? In what ways would the Soar architecture have to change? One hypothetical solution would be to add an additional rule learning mechanism to Soar that acts similarly to the production compilation mechanic of ACT-R. But instead of compiling rules from sequential rules the way ACT-R does, Soar might compile chunks learned from *indifferent operators*. That is, compile operators that have indifferent preference with respect to each other, such that it is known that they could be selected in any order. Compiling operators together would mean compiling both proposal and apply rules together, because an operator is defined by both. With gradual learning, this would mean that the agent could propose many different action line operations, select them in random order, and still register each possible pair as experienced together and as a candidate for compilation. This would allow a PROPs agent to represent action lines as a flat set in declarative memory and still achieve the desired levels of transfer. Any operator instruction would only lead to a single action context that contained all action lines together for the instructed behavior. The agent would learn the action line hierarchy implicitly like Actransfer, only by adding new procedural knowledge, not explicitly with a declarative hierarchy the way $PROP_3$ does.

Right now, it would seem that neither ACT-R nor Soar are able to fully support gradual PRIMs or PROPs learning in such a way that the agent learns the hierarchy for optimal transfer on its own. Each architecture has its own strengths and weaknesses for defining different parts of the PRIMs processing pipeline. Specifically, this work lets me conclude that:

- Soar supports P2-P3 in PRIMs theory through explicitly hierarchical instructions with parallel condition evaluation, which is not easily compatible with ACT-R's approach to goals, decision making, and WM.
- ACT-R supports P5 in PRIMs theory through implicitly hierarchical gradual, bottom-up compilation of practiced actions, which is not directly compatible with Soar's current approach to chunking and decision making.

Future work should investigate possible computational solutions in greater detail.

C.6 P6: Problem-Space Latency

How does the agent's cognitive processing with PRIMs map to temporal costs in task behavior to allow comparison with humans?

I described PROP$_3$'s simple approach for P6 with T_{ps} in section section 9.2. I here discuss this approach further in the context of cognitive architecture research.

In many cognitive architectures, including ACT-R and Soar, it is standard to treat the number of decision cycles required to carry out a task as a main factor of task behavior latency. The default in ACT-R is to assign 50 msec per decision cycle. This has often matched well with human data (Stewart & Eliasmith, 2009). ACT-R includes many more factors that can contribute to timing, such as time to access each of the various buffer modules that the architecture defines. Soar does not have a built-in function for mapping processing to timing, but researchers tend to follow the same conventions as used with ACT-R. With either ACT-R or Soar, the agent can demonstrate learning by requiring fewer and fewer decision cycles to perform a task, as a result of learning more efficient production rules.

In the architectural design of Soar or ACT-R, all procedural knowledge has full access to match on WM with no additional temporal cost. In PRIMs theory, however, unless `Auto` is enabled, rules cannot carry out task reasoning unless a corresponding declarative instruction has been retrieved from LTDM. This effectively implies additional time for procedural knowledge access. In PROP$_3$, the time required to load a procedure context in WM represents the latency for accessing the associated procedures.

Currently, Soar lacks a concrete theory for the latency of LTDM retrievals, beyond the two cycles required to query and retrieve a memory. A PROPs model is thus still incomplete when it comes to simulating performance in declarative memory retrieval tasks. However, declarative fact retrieval timing is not part of PRIMs theory as described in (Taatgen, 2013), and only retrievals for PRIMs instructions are strictly within the scope of the flow diagram of Figure 4.1. Further, the tasks examined in this research are dominantly procedural and WM tasks, and do not involve many non-instruction retrievals.

T_{ps} is separate from the time required to select and apply task operators as well as from the time required for the architecture to retrieve any facts from LTDM that are not procedure contexts.

Task set theory also treats the time to load or switch task sets as a distinct latency factor. As written in (Sakai, 2008): "To adopt a task set is to select, link, and configure the elements of cognitive processes necessary to accomplish the task. Thus establishing a task set is time consuming because it requires higher-order neural interactions between regions in the prefrontal and posterior association cortices that represent the elements of the task." "The time needed to establish task sets may also be one of the components that produce switch costs, which reflect the difference in performance between trials immediately after a task switch and trials that repeat the same task."

It should also be noted that retrieving different declarative or episodic memories in the task set literature is sometimes considered in terms of switching or loading task sets (Sakai, 2008). If this is the case, and if procedure contexts are a decent model of task sets, then this implies that one

might be able to extend the procedure context model to include a model of more classic declarative memory access. Something like a procedure context network in LTDM, call it a declarative context network, could represent a subject's semantic knowledge about the world, and the number of processing cycles it takes for the subject to navigate the network to retrieve the desired memory would model the same thing that $T_{retrieve}$ models. In this way of modeling, a declarative memory would be retrieved more quickly because it was more readily accessible through network links rather than because of an activation value. But link-distance and activation might, in fact, be two aspects of the same phenomenon. The retrieved memory might have a higher activation because of a smaller link distance from the current WM context. And if a declarative context has multiple outgoing links that the agent could select, it would make sense that the agent would prefer to follow the one with the higher activation, and thus be more quick to access the contexts pointed to by that link. These are open questions for further research.