Combining Procedural and Declarative Knowledge in a Graphical Architecture

Paul S. Rosenbloom (Rosenbloom@USC.Edu)
Department of Computer Science and Institute for Creative Technologies, 13274 Fiji Way
Marina del Rey, CA 90292 USA

Abstract

A prototypical cognitive architecture defines a memory architecture embodying forms of both procedural and declarative memory, plus their interaction. Reengineering such a dual architecture on a common foundation of graphical models enables a better understanding of both the substantial commonalities between procedural and declarative memory and the subtle differences that endow each with its own special character. It also opens the way towards blended capabilities that go beyond existing architectural memories.

Keywords:

Cognitive architecture; memory; graphical models; procedural; declarative; semantic; episodic; rules; constraints.

The distinction between procedural and declarative knowledge plays a central role in many cognitive architectures. ACT-R has long embodied distinct rule-based procedural and fact-based declarative long-term memories (Anderson, 1993). Early work with Soar instead leveraged a single rule-based long-term memory to support both procedural and declarative knowledge, with rules directly encoding procedures while also providing access paths to facts stored in their actions (Rosenbloom, Newell & Laird, 1991). Yet, Soar 9 has now followed ACT-R's lead, and in fact gone beyond it with distinct declarative memories for semantic and episodic knowledge (Laird, 2008). CLARION embodies the distinction in two different manners (Sun, 2006). It has a procedural Action Control System for controlling action and a declarative Non-Action Control System for general knowledge, but it also has a crosscutting distinction between explicit and implicit knowledge that applies to both of these modules and the whole architecture.

As part of an effort to investigate whether the potential of graphical models (Koller & Friedman, 2009) to unify signal, probability and symbol processing will enable development of simpler yet broader architectures than are seen today (Rosenbloom, 2009a), a new memory architecture with both procedural and declarative memories – but as yet without learning – has been implemented via a common graphical substrate. Guided by the functionality embodied in ACT-R's and Soar 9's long-term memories, the hopes for this implementation were to (1) achieve a straightforward mapping of these disparate memories onto the substrate, resulting in (2) a simpler and more uniform memory architecture, (3) embodying a blended functionality that can (4) exceed existing memory capabilities. The goal was not to model specific results from human memory research, but to understand the implications of graphical implementation and unification on such memory architectures.

Results to date have yielded a new blended memory architecture that is of interest for both the commonality among these memories that it leverages and the subtle differences among them that it exposes. The differences get at some of the most fundamental distinctions between procedural and declarative knowledge while continuing to drive research on their further unification. The next three sections describe the implemented memory architecture along with the commonalities it leverages; the differences this architecture reveals between procedural and declarative memory, as well as, as a bonus, those among different flavors of declarative memory; and what has been yielded so far in terms of blended functionality and new capability. The final section summarizes and looks to the future.

Memory Architecture

ACT-R and Soar 9 each embodies a procedural memory for rules plus a declarative (semantic) memory for facts. Soar 9 also goes a step further, implementing a second distinct declarative (episodic) memory for past history. Although ACT-R does not implement a separate episodic memory, there is work on how its existing mechanisms can yield comparable behavior (Sims & Gray, 2004). The focus here is on uniformly implementing all three of these long-term memory functionalities – one procedural and two declarative – via a common graphical substrate.

The memory architecture is built on top of a *graph layer* based on factor graphs and the summary product algorithm (Kschischang, Frey & Loeliger, 2001). Factor graphs are varieties of graphical models, like Bayesian networks, but enabling efficient computation with arbitrary multivariate functions by decomposing them into products of simpler subfunctions when suitable forms of independence exist; e.g., F(a,b,c) might decompose to $F_1(a,b)F_2(b,c)$. The reduced computation then maps to a bipartite graph in which there are *variable nodes* for variables and *factor nodes* for subfunctions (Figure 1). A variable node is linked to a factor node when the former's variable is used by the latter's function. The summary product algorithm passes messages along these links until quiescence is reached, with each message providing information about the possible values of the variable on the link. Each node computes its output messages by combining its incoming messages, plus its function if it is a factor node. The result is an inherently local computational model that can compute global results

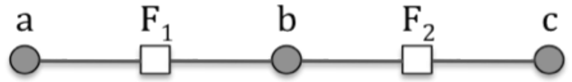

Figure 1: Factor graph for $F(a,b,c)=F_1(a,b)F_2(b,c)$.

across the cycles of message passing leading to quiescence, and that leverages independence for efficiency. It bears a relationship to neural networks, but combines additional breadth in some areas with more constraint in others.

The summary product algorithm is most often used to compute variable *marginals*, integrating information from across the graph to determine which values are legal, and what weights or probabilities are associated with them. When computing marginals, the algorithm typically uses *sum* for summarization, yielding the sum-product variant. When it is preferable to compute the *maximum a posteriori (MAP) estimation* – that is, the single most likely combination of values over all of the variables – *max* is used instead, yielding max-product. The graph layer here defaults to marginals (and sum), but can also compute MAP estimations and employ max when appropriate.

This graph layer is a reimplementation of the one developed in (Rosenbloom, 2009a) for rule match, with improvements in functionality, generality, and efficiency. The biggest change generalizes the representation for factor functions and messages from N dimensional Boolean arrays to N dimensional continuous functions (approximated as piecewise linear functions over rectilinear regions, as in Figure 2). Instead of just supporting symbol processing, this representation has the potential to support: continuous information for perception, imagery, and motor control; discrete distributions for uncertain information; and symbols for general reasoning. Starting from the continuous base, discrete distributions require discretizing variable domains; for example, breaking up the real line into unit segments, one per integer. Symbols then arise when the ranges of discrete variables are restricted to 0/1. A symbol table has also been added to map between unit segments and arbitrary symbols, but it is only for ease of programming and has no effect on the workings of the summary product algorithm.

$y \backslash x$	[0,10>	[10,25>	[25,50>
[0,5>	0	$.2y$	0
[5,15>	$.5x$	1	$.1+.2x+.4y$

Figure 2: Example (2D) piecewise linear function.

To implement the memory architecture, a *memory layer* was built on top of the graph layer that reifies a distinction between long-term and working memory, as in both ACT-R and Soar 9. Long-term memory structures compile into subgraphs that both store and access the knowledge. Working memory compiles into functions in peripheral factor nodes that remain fixed within a single cycle of memory access – i.e., within a single settling of the graph – but can be altered between cycles.

Long-term memory structures are specified at the memory layer as *conditionals*, generalized rules combining *patterns* and a *function*. Each pattern has a predicate plus one or more arguments specifiable as constants or variables; e.g., `Object(`*s*`,O1)` is a pattern with predicate `Object` plus the variable *s* (for states) and the constant `O1` (an object) as arguments. A pattern compiles into a linear graph structure that has a working-memory node at one end, a variable node at the other (for legal values of the pattern's variables), and factors that test pattern constants in between. This fragment corresponds to part of an *alpha network* in the Rete match algorithm, with the variable node acting as an *alpha memory* (Forgy, 1982). The big difference though is that in Rete messages always flow from working memory to the alpha memory. Here, messages can flow in either or both directions. As in Rete, the flow is away from working memory for *conditions* (Figure 3), but the flow is towards working memory for *actions*. *Condacts* – a neologism for *cond*itions and *act*ion*s* – are patterns for which the flow is bidirectional. A single conditional can have any combination of conditions, actions and condacts.

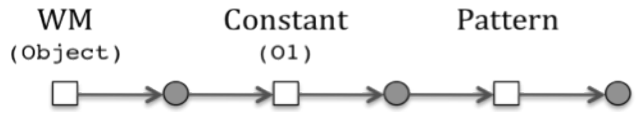

Figure 3: Alpha network for condition `Object(`*s*`,O1)`.

Patterns are combined into conditionals by a network of factor nodes that test equality of variable binding across patterns, plus variable nodes that represent combinations of variables across patterns. This portion of the factor graph corresponds to Rete's *beta network*, in which partial instantiations are joined to yield full rule matches. However, here the beta network connects conditions, actions, and condacts though bidirectional message flow.

Functions, when included, are defined over condact variables, and lead to new factor nodes that link with these variables. Functions can represent probability distributions over the cross products of the domains of condact variables, as is typical in many graphical models, but they also can represent other numeric and Boolean functions.

The conditional in Figure 4 uses a condition, a condact, and a function to define a prior distribution over the concept associated with object `O1` in the current state. Object `O1` can be a walker, a table, a dog or a person, each with its own prior probability. The variable in square brackets (*α1*) is a *pattern variable*. When multiple patterns, possibly across multiple conditionals, share a pattern variable, they compile to the same variable node within the graph. This enables chaining and local bidirectional communication among

```
CONDITIONAL ConditionPrior
   Condition: Object(s,O1)
   Condact: Concept(O1,c) [α1]
```

Walker	Table	Dog	Human
.1	.3	.5	.1

Figure 4: Concept prior over object O1.

conditionals within a single cycle of memory access, for, among other things, correct probabilistic reasoning. The factor graph for this conditional can be seen in Figure 5. Messages spread out from all nodes in this graph, following the directionality of the arrows. The primary constraint on this computation stems from local factor node functions, but as messages propagate, these constraints propagate as well.

If a conditional just has conditions and actions, it is a rule, and can form the basis for a traditional procedural long-term memory. Figure 6 shows a conditional defining a simple rule that performs a transitive computation. As with the earlier graph layer, match time per rule here has a worst-case bound that is exponential in the treewidth of the rule rather than the number of conditions.

```
CONDITIONAL Transitive
   Condition: Next(a,b)
              Next(b,c)
   Action: Next(a,c)
```

Figure 6: Transitive rule.

If a conditional only has condacts, we have symmetric flow among all of its patterns, and the basis for a declarative memory. Figure 7 shows a conditional (including part of the function) for a distribution over the weight of object `O1` given its concept. The `Concept` condact compiles to the same graph node created for it in conditional `ConditionPrior` (Figure 4). The function partitions the weight (in pounds) into a finite number of classes and assigns a linear function to each rectangular region defined by the cross product of the weight class and the concept.

A rule-based procedural memory consists of condition-action conditionals, such as the one in Figure 6. Given just this rule, the graph contains 7 factor nodes and 9 variable nodes. Match requires 47 messages to complete irrespective of the number of matching elements, since each message includes information about all matches. However, more matches may mean more calculation per message, yielding 5 ms elapsed time for one match and 16 ms for two.

A semantic memory implemented along the lines of Anderson's (1990) analysis of categorization and feature prediction includes conditionals for prior probabilities of concepts – such as the one in Figure 4 (although possibly without the condition) – and conditional probabilities of object attributes given concepts, as in Figure 7. By linking these conditionals through the concept's pattern variable, an object's cued features can yield a posterior distribution over its concept – based on conditional probabilities of cued

```
CONDITIONAL ConceptWeight
   Condact: Concept(O1,c)[α1]
            Weight(O1,w)[α2]
```

w\c	Walker	Table	…
[1,10>	.01*w*	.001*w*	…
[10,20>	.2-.01*w*	"	…
[20,50>	0	.025-.00025*w*	…
[50,100>	"	"	…

Figure 7: Conditional probability of weight given concept.

features plus the prior probability of the concept – and this posterior concept distribution can then combine with the conditional probabilities of uncued features to generate probabilistic predictions of their values, all within a single memory cycle. In the particular example used, in addition to the continuous weight feature, there is one discrete numeric feature (legs) plus three symbolic features (color, alive, mobile). The graph comprises 47 factor nodes and 47 variable nodes. Given the cue that the color is silver, quiescence is reached after 634 messages, requiring 100 ms. It predicts that the concept is *walker* because almost all walkers are silver while only a small fraction of dogs and tables are. It also predicts that the cued object is mobile, not alive, has four legs and weighs 10 pounds.

In Soar 9, episodic memory retrieves the most recent episode that best matches the cue, effectively acting as a temporal instance-based semantic memory. This can be implemented much like semantic memory, but with alterations for recency and for retrieving the single best episode given a cue rather than predicting the most likely features given the cue. For recency, a discrete temporal variable replaces the concept variable, with a prior distribution that tails off exponentially into the past (Figure 8). To retrieve the single best episode, each feature conditional specifies the conditional probability of its values over the past history, and shares the `Time` condact with the temporal prior (Figure 9). The implemented example uses the same features as the semantic memory, but stores an object instance at each time step. The graph has 46 factor nodes and 46 variable nodes. Given the cue that the concept is human, it takes 433 messages, over 35 ms, to select the more recent of the two humans seen (at time step 3).

The straightforward implementation of these three varieties of long-term memory via the memory layer goes a long way towards realizing the first hope stated up front. In

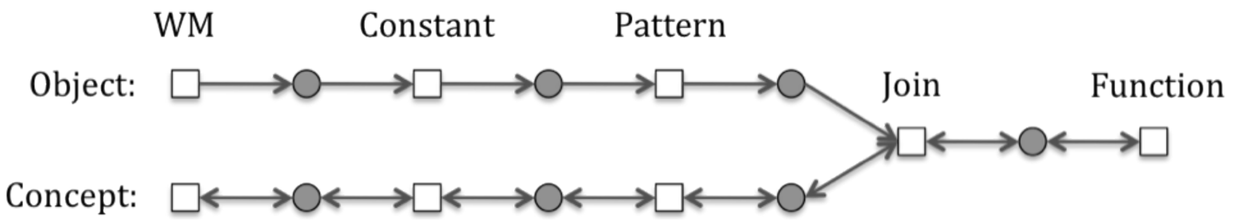

Figure 5: Factor graph for conditional in Figure 4, with a condition (`Object`), a condact (`Concept`), and a function.

```
CONDITIONAL TimePrior
    Condact: Time(t) [α3]
```

0	1	2	3	4
0	.032	.087	.237	.644

Figure 8: Exponentially decaying, discrete, temporal prior.

```
CONDITIONAL TimeConcept
    Condact: Time(t) [α3]
             Concept(O1,c)
```

t\c	Walker	Table	Dog	Human
1	1	0	0	0
2	0	0	0	1
3	0	0	0	1
4	0	0	1	0

Figure 9: Conditional probability of concept given time.

comparison to the earlier implementation of just a rule-based procedural memory, there is additional complexity here in extending rules to conditionals, and in moving from symbolic to continuous values, but then very little more is needed to implement these particular variants of procedural, semantic and episodic memories. With respect to the second hope's appeal to simplicity and uniformity, there is indeed much in common across the implementations of these three memories: they all build on the distinction between working memory and long-term memory; long-term memory is uniformly represented as conditionals that compile into factor graphs, while working memory is encoded as evidence in peripheral factor nodes; and memory access is cued by working memory through the application of the summary product algorithm to the resulting graph.

One difference of note between this implementation and Soar 9 arises from Soar's ability to perform multiple cycles of procedural (rule) access within a single decision cycle, but only one cycle of declarative (semantic or episodic) access. The memory architecture here is instead limited to just one cycle of memory access per decision cycle for both declarative and procedural knowledge. In (Rosenbloom, 2009b), early experiments with graphical models led to the hypothesis that global computation in Soar should only happen over a full decision cycle rather than once per rule cycle, and that Soar was thus inconsistent in allowing global access to working memory each rule cycle. The current implementation abides by this constraint; however, using pattern variables still allows chaining of rules within a decision cycle, but now based on local communication between actions of earlier rules and conditions of later ones.

Differences

The most obvious difference between the implementations of procedural and declarative memory is the use of conditions and actions in procedural memory versus condacts in declarative memory. At the graph level this reduces simply to the directionality of information flow in the alpha networks, but it does yield a qualitative difference at the memory level. With unidirectional information flow, rules predefine what are to be the cues for retrieval (conditions) and what is to be retrieved (actions). This is particularly effective for procedures as it enables directional if-then programming. In contrast, with bidirectional information flow, both varieties of declarative memory dynamically determine at access time what aspects of an object are cues and therefore what aspects are to be retrieved (i.e., those aspects not cued). This significantly enhances the flexibility of access, but eliminates the directionality that is exploited in procedural programming.

A more subtle difference is whether a *closed world* or *open world* assumption occurs with respect to working memory. Rule-based systems use the former, assuming that anything not in working memory is false. The use of negated conditions depends on this assumption, as does the ability to keep working memory small and focused. On the other hand, declarative memories – and most logical and probabilistic models – use an open world assumption, that the truth of anything not explicitly in evidence is unknown. This enables values that are unknown prior to memory access to be retrieved/predicted by condacts during such access. With a closed-world assumption, this becomes impossible because any values not explicitly true prior to access would be set to false, leading to a conflict with any attempt to make a positive predication during access. Rules avoid this problem because their retrievals/predictions occur non-monotonically at the end of the access cycle, by actions that don't examine working memory during the cycle.

This difference is realized in the graph layer by declaring individual predicates to be closed or open world when they are defined; an idea adopted, along with the use of predicates, from earlier experiments with Markov logic (Domingos & Lowd, 2009) as a general *implementation level* for architectures (Rosenbloom, 2009b). Closed-world predicates are primarily used in conditions and actions and open-world predicates in condacts.

A third difference concerns whether memory access retrieves all cued results or only the best result. In Soar 9's rule-based procedural memory, all combinations of bindings of condition variables to working memory constants yield rule instantiations that fire in parallel. In contrast, cuing of either semantic or episodic memory should return only the best result. At the graph layer, this difference is interpreted in terms of distinct types of variable domains. When only the best result is desired, the variable's domain is declared *unique*, and messages about it are normalized to sum to 1. This yields a distribution over the variable's domain elements for the probabilities that they are to be retrieved. When all results are to be returned, the variable domain is declared to be *multiple*, and its messages are not normalized. In such cases, each domain element acts roughly as its own Boolean variable, with a value of 1 if it is to be retrieved and 0 otherwise; thus encoding all bindings of the variable in each message. The summary product implementation then uses max to summarize over *multiple* variables, even when marginalizing, bounding the result above by 1.

These three differences – (1) the directionality of information flow in alpha networks, (2) a closed-world versus open-world assumption, and (3) unique versus multiple variables – jointly distinguish procedural from declarative memories in this implementation. Of these, the first appears to be the most fundamental, to the point where it justifies an explicit hypothesis that such a difference will always be found in comparing procedural and declarative memories. The other two are less clear. It may be possible, for example, to build an effective procedural memory based on an open-world assumption. If so, the second difference would not then be essential. Likewise, if an effective procedural memory can be based on returning only the best result – more like how rules work in ACT-R than in Soar 9 – the third difference may not be essential.

In addition to the differences just identified between procedural and declarative memory, three differences of note also showed up between the two implemented flavors of declarative memory: semantic and episodic. First, semantic memory searches for the most likely value for each attribute of an object individually – by marginalizing via sum-product – while episodic memory instead computes MAP estimation via max-product to retrieve the most appropriate single episode (where all of an episode's attributes jointly contribute to determining its appropriateness). Second, the probabilities of features in semantic memory are conditional on the concept while in episodic memory they are conditional on the time. Third, semantic memory is based on a general probabilistic representation of the values of attributes (see Figure 7), while episodic memory is based on the history of specific instances actually experienced (see Figure 9).

As with the differences between procedural and declarative memory, the first difference here appears to be fundamental, at least given this form of semantic memory. The other two differences appear less fundamental. It is possible, for example, to implement an instance-based semantic memory where the concept is just another feature. Sum-product can then dynamically compute more general feature distributions by summarizing over these instances. Interestingly, when max-product is used instead, the individual object that best matches the cues is retrieved, yielding something more like the semantic memory implemented in Soar 9. One intriguing implication is that the causative difference between generalization and analogy/CBR/nearest-neighbor may reduce to whether sum-product or max-product is used over an instance-based memory. The former generalizes over all instances, while the latter retrieves the single best instance.

Blended Functionality and New Capabilities

Beyond the three memories implemented above, the flexibility of the conditional representation enables blending of functionality across these memories (hope three) plus new capabilities beyond them (hope four). Blending arises from the flexibility with which conditions, actions, condacts and functions can combine within individual conditionals, plus the flexibility with which multiple conditionals can interact within long-term memory.

Conditionals by themselves enable combining procedural and declarative functionality within individual memory units. Semantic memory provides a good example. In addition to condacts and a function, each conditional can also include a condition that matches multiple objects in working memory. The prior is then represented by a conditional similar to the one in Figure 4, but with the constant `O1` replaced by a variable. The individual feature conditionals then resemble Figure 7, but with the condition added and the variable substituted (Figure 10). Like Soar 9, there is still a limit of one cycle of semantic memory retrieval per cycle of memory access – if quiescence of message passing in summary product is mapped onto quiescence of rule firing in Soar 9 – but unlike Soar 9, features of many objects can be predicted in parallel within this single cycle of memory access.

```
CONDITIONAL ConceptWeightGeneral
   Condition: Object(s,o)[α4]
   Condact: Concept(o,c)[α5]
            Weight(o,w)[α6]
```

Figure 10: Conditional distribution for semantic memory with condition to match objects (shown without function).

Other forms of within-conditional blends are also possible, such as combining conditions, actions and functions to yield weighted rules. Beyond this, to blend functionality across conditionals requires communication across conditionals that nominally belong to different memories, either via pattern variables within a single cycle of memory access or through working memory across cycles. The rule in Figure 11, for example, uses pattern variables to access the results of Figure 7's semantic retrieval, and generates a new `ConceptWeight` predicate. This also exploits within-conditional blending, but here in service of across-memory interaction.

```
CONDITIONAL ConceptWeightRule
   Condition: Object(s,o)[α4]
   Condact: Concept(o,c)[α5]
            Weight(o,w)[α6]
   Action: ConceptWeight(c,w)
```

Figure 11: Accessing semantic memory results in a rule.

Further work will be required to fully understand the range of capabilities this memory architecture might yield, and what the implications might then be for cognitive modeling. But at least one major new memory capability – for *constraints* – has already become apparent. Constraints are structures that specify restrictions on values assigned to variables (Dechter, 2003). Given a set of variables with well-defined domains, and a set of constraints over these variables, constraint satisfaction determines which combinations of domain values are consistent with the constraints. Constraints are like rules in yielding all combinations of variable bindings, but like declarative memory in their flexibility of access, and thus in their use of condacts and an open world assumption. Figure 12 shows a constraint for the two-color problem, implemented via condacts and a Boolean function. The pattern variables for the two regions are shared with other constraints over those regions to enable appropriate propagation over the whole network during message passing. Although not a common form of long-term memory in cognitive architectures, except in neural systems based on "soft" constraints (Ackley, Sejnowski & Hinton, 1985), constraints do play a significant role in a variety of AI systems and languages.

```
CONDITIONAL TwoColorConstraint12
   Condact: Color(R1,c1)[α7]
            Color(R2,c2)[α8]
```

c1\c2	Red	Blue
Red	0	1
Blue	1	0

Figure 12: Two-color constraint between regions R1 & R2.

Summary

Basing a memory architecture on the uniform breadth of graphical models has enabled straightforward construction of four distinct memories: a rule-based procedural memory, semantic and episodic declarative memories, and a constraint memory that is functionally a hybrid between the two. These implementations reveal significant commonality among these memories, but also subtle differences. Of the differences, unidirectional versus bidirectional message passing appears to be most fundamental when comparing procedural and declarative memories, while marginalization versus MAP estimation appears to be most fundamental when comparing semantic and episodic memory.

Implementing memories in this manner also enables blending capabilities across memories and creating new unanticipated kinds of memories, such as a constraint memory. This general approach holds the promise of extending beyond memory architecture to full cognitive architectures with mechanisms for decisions, learning, and perceptuomotor behavior. The hopes for this larger effort would be to derive a better understanding of: the diverse mechanisms involved, including their commonalities and differences; how they can and should work together; and how to go beyond the kinds of combinations currently seen to simpler yet more comprehensive cognitive architectures.

Acknowledgements

This effort has been sponsored by the USC Institute for Creative Technologies and the U.S. Army Research, Development, and Engineering Command (RDECOM). Statements and opinions expressed do not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred. I would like to thank Bill Swartout for help in restructuring this work for publication.

References

Ackley, D. H., Hinton, G. E. & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, *9*, 147-169.

Anderson, J. R. (1990). *The Adaptive Character of Thought*. Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1993). *Rules of the Mind.* Erlbaum.

Dechter, R. 2003. *Constraint Processing*. San Francisco, CA: Morgan Kaufmann.

Domingos, P. & Lowd, D. (2009). *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool.

Forgy, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, *19*, 17-37.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA: MIT Press.

Kschischang, F. R., Frey, B. J. & Loeliger, H. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, *47*, 498-519.

Laird, J. E. (2008). Extending the Soar cognitive architecture. *Artificial General Intelligence 2008: Proceedings of the First AGI Conference*. IOS Press.

Rosenbloom, P. S. (2009a). Towards a new cognitive hourglass: Uniform implementation of cognitive architecture via factor graphs. *Proceedings of the 9th International Conference on Cognitive Modeling*.

Rosenbloom, P. S. (2009b). A graphical rethinking of the cognitive inner loop. *Proceedings of The IJCAI International Workshop on Graph Structures for Knowledge Representation and Reasoning*.

Rosenbloom, P. S., Newell, A. & Laird, J. E. (1991). Towards the knowledge level in Soar: The role of the architecture in the use of knowledge. In K. VanLehn (Ed.), *Architectures for Intelligence*. Hillsdale, NJ: Erlbaum.

Sims, C. R. & Gray, W. D. (2004). Episodic versus semantic memory: An exploration of models of memory decay in the serial attention paradigm. *Proceedings of the 6th International Conference on Cognitive Modeling* (pp. 279-284).

Sun, R. (2006). The CLARION cognitive architecture: Extending cognitive modeling to social simulation. In R. Sun (Ed.), *Cognition and Multi-Agent Interaction*. New York, NY: Cambridge University Press.