Journal of Artificial General Intelligence 7(1) 1-103, 2016
DOI: 10.1515/jagi-2016-0001

Submitted 2015-10-01
Accepted 2016-6-21

The Sigma Cognitive Architecture and System: Towards Functionally Elegant Grand Unification

Paul S. Rosenbloom ROSENBLOOM@USC.EDU
Abram Demski ADEMSKI@ICT.USC.EDU
Institute for Creative Technologies and
Department of Computer Science
12015 Waterfront Dr.
Playa Vista, CA 90094, USA

Volkan Ustun USTUN@ICT.USC.EDU
Institute for Creative Technologies
12015 Waterfront Dr.
Playa Vista, CA 90094, USA

Reviewers: Ahmad Al-Kashef, Roland Fernandez, Ho John Lee, Timothy Mann, Javier Snaider, Mary-Anne Williams

Editor: Brandon Rohrer

Abstract

Sigma (Σ) is a cognitive architecture and system whose development is driven by a combination of four desiderata: *grand unification*, *generic cognition*, *functional elegance*, and *sufficient efficiency*. Work towards these desiderata is guided by the *graphical architecture hypothesis*, that key to progress on them is combining what has been learned from over three decades' worth of separate work on *cognitive architectures* and *graphical models*. In this article, these four desiderata are motivated and explained, and then combined with the graphical architecture hypothesis to yield a rationale for the development of Sigma. The current state of the cognitive architecture is then introduced in detail, along with the graphical architecture that sits below it and implements it. Progress in extending Sigma beyond these architectures and towards a full cognitive system is then detailed in terms of both a systematic set of higher level *cognitive idioms* that have been developed and several *virtual humans* that are built from combinations of these idioms. Sigma as a whole is then analyzed in terms of how well the progress to date satisfies the desiderata. This article thus provides the first full motivation, presentation and analysis of Sigma, along with a diversity of more specific results that have been generated during its development.

Keywords: Cognitive architecture, graphical models, cognitive system, Sigma

1. Introduction

Sigma (Σ) is a nascent cognitive system, the beginnings of an integrated computational model of intelligent behavior, that is built around an eponymous cognitive architecture: a model, which is also called Sigma, of the fixed structure underlying a cognitive system (Langley, Laird, and Rogers, 2009). It harks back to the original grand goal of artificial intelligence (AI), as well as to the more recent reformulation of this goal within artificial general intelligence (AGI) (Goertzel, 2014), in seeking a working implementation of a full cognitive system. As such, it is intended ultimately to support the real-time needs of intelligent agents, robots and virtual humans.

Sigma also has roots in the grand goal within cognitive psychology/science of developing a unified theory of human cognition (Newell, 1990; Anderson, 2007). The desire with Sigma is for a fully working and useful system whose development is guided heuristically by what is known about human cognition, and which will also ultimately explain human intelligence at some appropriate level of abstraction. In service of this goal, maintenance of a fully working system at all times has been essential, with short-term gains in the form of detailed matches to human behavior sacrificed as necessary in return for long-term gains in functionality and integration, but with the ultimate hope that human cognition will be explainable in terms of the capabilities provided plus the interactions, both positive and negative, among them.

Both Sigma's overall development path and its relatively unique niche within cognitive systems/architectures have been driven by a quartet of general desiderata: (1) *grand unification*, spanning not only traditional cognitive capabilities but also key non-cognitive aspects such as perception, motor control, and affect; (2) *generic cognition*, spanning both natural and artificial cognition at a suitable level of abstraction; (3) *functional elegance*, yielding broad cognitive (and non-cognitive) functionality – ultimately all that is necessary for human(-level) intelligence – from a simple and theoretically elegant base; and (4) *sufficient efficiency*, executing quickly enough for real-time applications involving virtual humans and intelligent agents/robots, and for large-scale experiments in modeling human cognition. Three of these desiderata – grand unification, functional elegance and sufficient efficiency – have been explicit for some time in the development of Sigma. The new desideratum – generic cognition – has until now been largely implicit because of the predominant initial focus on functionality. But, given that Sigma's development has always been guided by an understanding of both natural and artificial cognition, and that a unified theory of cognition has always been an important long-term goal of work in Sigma, expanding the earlier trio to a full quartet seemed both necessary and appropriate.

These desiderata express the long-term goals we are working towards with Sigma, while not necessarily making claims about what Sigma currently achieves. The body of this article is about what currently exists – everything discussed is actually implemented and running as part of an integrated architecture unless otherwise noted – but Sigma still does fall short in significant ways of the full desiderata. To better understand this gap, a qualitative assessment of how well Sigma currently meets the desiderata, along with a discussion of where it concomitantly is known to fall short, will be attempted. Cognitive architectures and systems are notoriously difficult to evaluate via traditional experimental approaches, but sidestepping this to investigate how well such a system meets a challenging set of desiderata can yield an alternative approach with real value.

The goals of this article are to provide the first thorough explanation of the Sigma cognitive system in print, and perform the interim evaluation just mentioned of how well it currently satisfies the four desiderata. The remainder of this introduction, however, first lays important groundwork for what is to come by presenting the key hypothesis driving Sigma's approach to achieving its four desiderata, identifying the research strategy used in developing Sigma and the range of results generated with it to date, overviewing Sigma's structure as a cognitive system, and laying out a map of the rest of this article.

1.1 Graphical Architecture Hypothesis

The essence of how Sigma is to meet its four desiderata is captured by the *graphical architecture hypothesis*; that is, the key at this point to achieving them is blending what has been learned from over three decades of independent work in *cognitive architectures* and *graphical models* (Pearl, 1988; Koller and Friedman, 2009). With respect to cognitive architectures, an effort has recently begun to capture the consensus lessons from the general history of cognitive architectures in the form of a *standard model* of them (Lebiere, 2013). General architectural lessons that have been leveraged in Sigma include the need for a long-term memory, a working memory, and perceptual buffers; the importance of multiple forms of long-term memory; the centrality of the ~50 msec cognitive cycle; and the criticality of combining symbolic and statistical information. Section 3 includes a discussion of additional, more specific lessons from particular architectures that have influenced the development of Sigma, while also making the case for why the development of a new architecture such as Sigma is necessary.

A technical introduction to graphical models is provided in Section 5.1, but in general they concern efficient computation with complex multivariate functions by leveraging forms of independence to decompose them into products of simpler functions, mapping the resulting decompositions onto graphs, and computing over these graphs via message passing or sampling algorithms. Graphical models provide working approaches to both symbol and signal processing, and to both logical and probabilistic reasoning, thus implying a potential to support the wide range of capabilities demanded by grand unification. They underlie much of modern artificial intelligence while also being closely related to a pair of dominant modern threads in cognitive modeling – neural networks (Jordan and Sejnowski, 2001) and probabilistic reasoning (Oaksford and Chater, 2007) – and being capable of producing classical symbolic models (Domingos and Lowd, 2009; Rosenbloom, 2009). Graphical models can, furthermore, yield all of this from a single form of representation and inference, such as factor graphs and the summary product algorithm (Kschischang et al., 2001), thus implying a potential to support functional elegance. They also can go beyond merely yielding algorithms for processing signals, probabilities and symbols, to often defining the state of the art, and thus plausibly enabling sufficient efficiency.

Graphical models have taken much of artificial intelligence and cognitive science by storm, while also making significant contributions in other fields (Kschischang et al., 2001). They have also inspired a spate of recent AI languages (Milch et al., 2007; McCallum et al., 2008; Domingos and Dowd, 2009). However, what they haven't done is have a significant impact on cognitive architectures and systems. Graphical models should enable a much broader and tighter integration of the capabilities required in a full cognitive system while supporting working systems and cognitive models embodying enhanced simplicity, elegance and efficiency.

Early work on Sigma occurred under the generic label of *a graphical cognitive architecture* (Rosenbloom, 2011a) to emphasize the essential graphical nature of the new work, and because it was ambiguous whether what was being developed was more a framework based on graphical models for exploring the space of cognitive architectures/systems or a specific architecture/system. A name was finally chosen when it could no longer be ignored that one particular graphical architecture/system was emerging. *Sigma* is a proper noun rather than an acronym, although it was inspired by "*S*igma is an *i*ntegrated (and/or *i*ntelligent) *g*raphical *m*odel *a*rchitecture." The name was also motivated by the integrative nature of Sigma (with the goal of summing across many capabilities), the core use of the *sum*(mary) product algorithm (Kschischang et al., 2001), and the metaphorical ambition to exceed the existing state of the art by at least a standard deviation.

The primary point of the graphical architecture hypothesis is to make explicit what is most unique about Sigma's approach to achieving the above desiderata rather than to propose a scientific hypothesis to be tested via controlled experiments. However, in a significant sense the entire Sigma research programme is a test of the hypothesis, which is validated to the extent that Sigma succeeds by leveraging it. As this paper proceeds, there will be numerous examples of the power of the hypothesis, along with some situations in which it is challenged but not quite invalidated. Together, the desiderata and the graphical architecture hypothesis provide the two legs on which Sigma now stands.

1.2 Research Strategy and Results

Given Sigma's desiderata and the graphical architecture hypothesis, the overall research strategy has been to investigate capabilities suggested by an amalgam of the disparate mechanisms and capabilities provided by existing cognitive architectures and systems, and how Sigma might provide these in a functionally elegant manner; further mechanisms and capabilities needed for grand unification; low hanging fruit, in terms of what graphical models most readily provide in terms of both functionality and modeling; and what is presently doable. For each new capability selected for investigation, functional elegance encourages deconstructing it in terms of existing mechanisms (Rosenbloom, 2012a, 2014) to the extent *feasible*, where feasibility is defined as (potential) sufficient efficiency; (potential) consistency with our understanding of human cognition at an appropriate level of abstraction; and reusability as a component in constructing even more complex capabilities (Rosenbloom, 2015). When architectural changes are required to provide a new capability, preference is given to minimal changes that can yield the new capability when combined with reuse of existing architectural mechanisms rather than to modular construction of the new capability from scratch.

This strategy contrasts with an approach in which a necessary and sufficient set of modules and their interconnections is hypothesized at the beginning, and then the bulk of the work consists of implementing and testing the modules, both individually and in combination. We do not claim to know at the beginning what modules will be necessary by the end. Instead, what we learn from each incremental step informs the decisions that come later about both what to implement and how to implement it. There is thus no overall architectural diagram of Sigma that shows all that is anticipated to be part of the architecture, with some boxes implemented and others still nominal. To a large extent, though, such a diagram is replaced by the desiderata as an expression of all that is ultimately to be there, and the evaluation of what does and does not exist at the moment. As will be discussed further, the non-modularity of Sigma's design that is encouraged by functional elegance also makes the creation of such an architectural diagram problematic.

Since 2008, when the development of Sigma was begun, progress towards grand unification has occurred across learning and memory (Rosenbloom, 2010, 2012a, 2014; Rosenbloom et al., 2013; Pynadath et al., 2014; Ustun et al., 2014; Ustun and Rosenbloom, 2015), perception and imagery (Chen et al., 2011; Rosenbloom, 2011b, 2012b; Joshi et al., 2014), reasoning and problem solving (Chen et al., 2011; Rosenbloom, 2011c), speech and language (Joshi et al., 2014; Rosenbloom et al., 2013; Ustun et al., 2014), and social and affective cognition (Pynadath et al., 2013, 2014; Rosenbloom, Gratch and Ustun, 2015). Insights from both natural and artificial cognition have been incorporated, and some novel contributions to cognitive modeling – such as the emergence of *base-level activation* as the prior learned over a temporal variable in episodic memory (Rosenbloom, 2014) – have started to appear. Providing so much generality and diversity in a functionally elegant and sufficiently efficient manner has been a constant challenge.

Functional elegance, for example, makes the development of new capabilities more a process of discovery than one of construction, while simultaneously constraining what is possible in terms of special purpose optimizations. However, functional elegance has also implied that any optimization to the general core that is provided by graphical models should yield pervasive speed improvements (Rosenbloom, 2012c; Rosenbloom, Demski and Ustun, 2015).

Listing this overall range of results does not imply that the underlying capabilities can necessarily be considered as fully complete within Sigma, but enough of each has been demonstrated to lay a plausible case, and to begin exploring interactions among them. These demonstrations also begin to lay the case that Sigma will ultimately be able to support the full behavioral arc from perception and understanding, up through reasoning and cognition, and back down to planning and action, incorporating in essence all of the computational aspects of intelligent behavior entirely within the system rather than via interfaces to external modules.

1.3 Overall Structure of Sigma as a Cognitive System

At a high level, Sigma as a cognitive system is best understood in terms of a sequence of layers that are analogous to those found in computer systems. Historically, the notion of a cognitive architecture was inspired by that of a computer architecture – not coincidentally, Allen Newell, the father of cognitive architecture, co-authored with Gordon Bell an early seminal text on computer architecture (Bell and Newell, 1971). Both forms of architecture are fixed structures that provide a language in which the system can be programmed; or, more appropriately for cognitive architectures, a language in which knowledge and skills can be embodied and learned. But, the analogy can also productively be extended beyond this single layer, as in Figure 1.

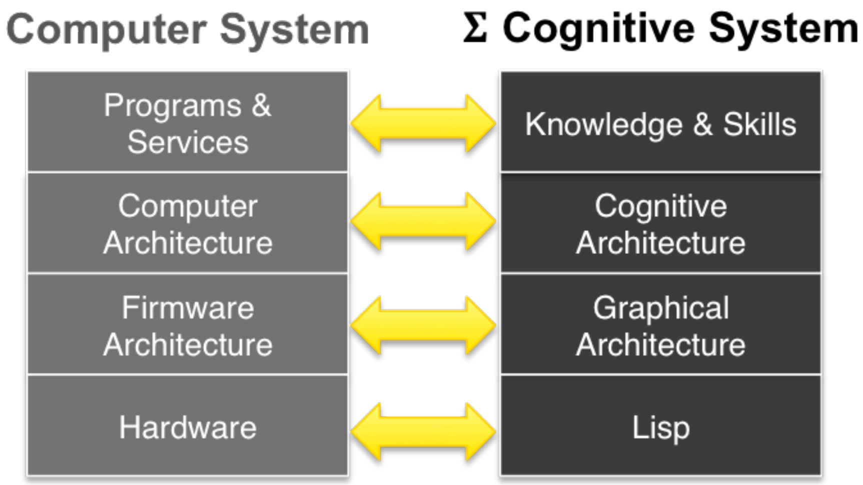


Figure 1: Corresponding computer and Sigma cognitive systems layers.

Below the computer architecture is a more parallel firmware architecture that defines a microcode language for use in bridging the gap between what the hardware directly provides at the layer below and the desired computer architecture. The firmware itself is not hardware, but must be present at boot time and is rarely if ever changed. This layer maps in Sigma onto a *graphical architecture*, a largely parallel lower layer based on factor graphs (Kschischang et al., 2001) – an undirected (bidirectional) form of graphical model – that bridges between Lisp, Sigma's analogue of computer hardware, and the cognitive architecture. The graphical architecture is the main algorithmic driver of behavior in Sigma, with constructs expressed within the cognitive architecture being compiled down to structures and functions at this level that can be directly processed. It is uncommon to think of multiple levels of architectures in cognitive systems, rather than just a single layer defining the cognitive architecture, but just as it can provide leverage in computer systems to have multiple architectural levels, each with its own distinct role and defining its own language, it can also be of value in cognitive systems.

Sigma's cognitive and graphical architectures form the two halves of the graphical architecture hypothesis, with the former capturing much that has been learned from Soar (Rosenbloom et al., 1993; Laird, 2012), ACT-R (Anderson et al., 2004), and other cognitive architectures, and the latter capturing the essence of what graphical models have to contribute. The dynamic tension, interplay and constraint between these two architectures, in both their development and execution, is the source of much of Sigma's unique contributions to cognitive architectures, artificial (general) intelligence, and cognitive science. Both architectures are intended to stay relatively simple and elegant, with straightforward compilation from the cognitive level to the graphical level. Much of the optimization effort within Sigma has concerned the efficiency of the graphical architecture (Rosenbloom, 2012c; Rosenbloom, Demski and Ustun, 2015), but this will not be a major focus in this article. The latest instantiation of Sigma – version 38 of Spring 2016 – comprises ~20K lines of code for these two architectures, including the compiler and ancillary code (such as an interactive graphical interface).

Above the computer architecture is a software layer that is home to a diversity of programs and services, and which itself can be decomposed into further sublayers that are home to such things as operating systems, programming languages and compilers. In Sigma this maps onto a similarly diverse layer of knowledge and skills. One fragment of this layer that is of particular importance in this article comprises *cognitive idioms*, which are effectively Sigma's analogues of software idioms, design patterns, libraries and services. Cognitive idioms provide conventional ways of structuring knowledge and skills to yield many of the capabilities typically provided directly as part of other cognitive architectures. For example, dividing long-term memory into procedural and declarative memories, and dividing declarative memory further into semantic and episodic memories, occurs via cognitive idioms rather than architectural modules in Sigma. Similarly, distinguishing concept learning from perceptual learning, and both of these from reinforcement learning, occurs via idioms. Being able to construct, integrate, and build hierarchically on a wide variety of cognitive idioms is crucial to the criterion of *feasibility* mentioned previously. It is also the source of many of Sigma's other unique contributions

The earlier work under the heading of *a graphical cognitive architecture* actually conflated the cognitive idioms from the top layer with the cognitive architecture at the layer below. This combination made sense at the time because many of the capabilities implemented in standard cognitive architectures map onto cognitive idioms in Sigma. So, thinking in terms of what exists in Sigma that is architectural in other systems led to this combination. However, extending the overall analogy from computer systems has helped tease out the appropriate distinction between an architecture (and its language) versus idiomatic usages of the language; and thus clarified that much of what is architectural in other systems is idiomatic in Sigma. This idiomatic provision of such capabilities is a core part of what makes Sigma functionally elegant, with the manner in which the graphical architecture implements the cognitive architecture providing the remainder.

1.4 This Article

This is admittedly a very long article, but its length enables us to describe both Sigma and the results generated with it to date in sufficient detail. The body of this article begins with a deeper discussion of Sigma's driving desiderata in Section 2, followed by a discussion in Section 3 of Sigma's raison d'être given these desiderata, the graphical architecture hypothesis, and the space of existing cognitive architectures. Readers most interested in the *why* of Sigma may want to focus on these two sections, while readers who care only about *how* it works and *what* it yields should feel free to skip them. Sections 4 and 5 detail the architecture, with the former covering the cognitive architecture and the latter the graphical architecture. (The lowermost Lisp layer, along with how the graphical architecture is implemented on top of it, is not the focus of this article, but interested readers can see Rosenbloom, 2012c; Rosenbloom, Demski and Ustun, 2015). The cognitive architecture is presented (almost completely) without appeal to the graphical architecture to enable those who are not interested in the lower level details of the graphical architecture to skip over them. Sections 4.4 and 5.4 also dig further into details than some readers may need, with the former describing the cognitive language and the latter the compiler that converts expressions in this language into graphs. Again, readers should feel free to skip them if so desired, although at least a cursory understanding of the cognitive language will be helpful in following the examples that come later.

Sections 6 and 7 present results, the former in terms of a set of capabilities and their associated idioms that can be yielded via knowledge and skills in the layer above the cognitive architecture, and the latter via two early-stage virtual humans that are each based on integrating multiple idioms. These sections, along with Section 4, are the right place to focus for readers who want to understand the scope of the capabilities Sigma can deploy at present.

The results of concern in this article comprise the two architectures and how they relate, the cognitive idioms and how they are deconstructed in terms of the cognitive architecture, and the two virtual humans. Based on these results, Section 8 qualitatively evaluates progress to date in Sigma by analyzing how well it presently satisfies its four desiderata, with Section 9 then wrapping up and summarizing what has been accomplished.

2. Desiderata

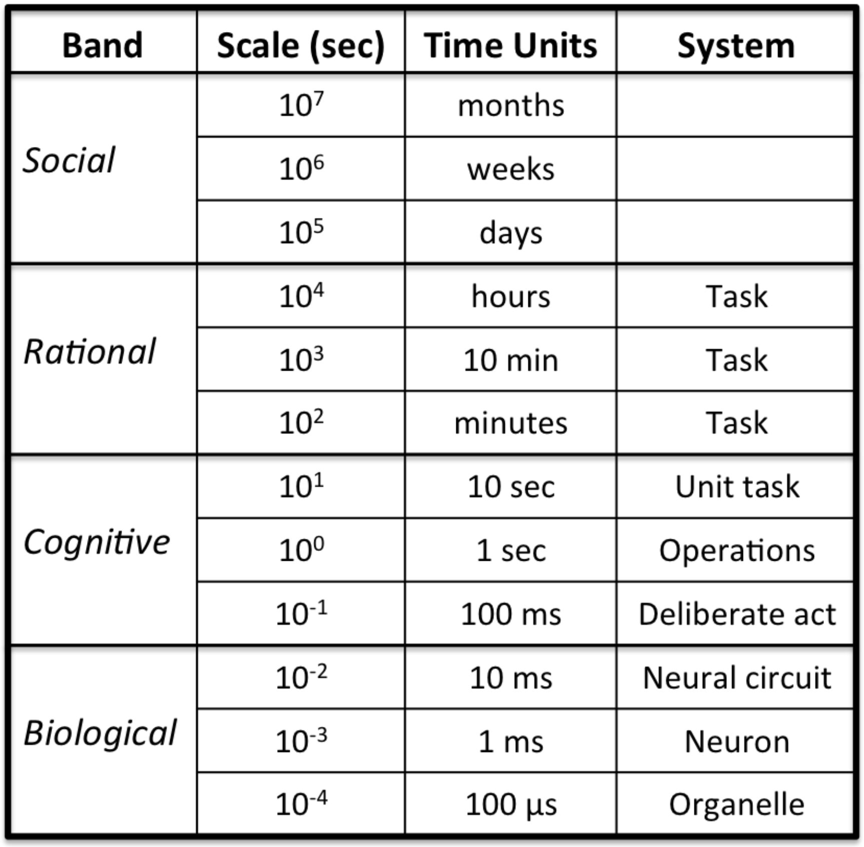
Most research is driven by desiderata that are an implicit part of the discipline's standards and norms. Research in computer science, for example, is driven by the desire to understand what computers are capable of, plus the achievement of key properties – such as tractability, efficiency, robustness, and security – concerning how these capabilities can be provided. As a subdiscipline within computer science, artificial intelligence narrows this desire to those capabilities hypothesized to be necessary for intelligent behavior, and where the key properties tend to focus on the quality of the outputs generated – such as how accurately a learned classifier categorizes previously unseen inputs – and the tractability or efficiency of the requisite processing. Research in cognitive science is instead driven by the desire to understand how people think, with experiments that illuminate aspects of this process and models that attempt to accurately reproduce, and hopefully explain, the results.

Research in cognitive systems and architectures is driven by the desire to understand and/or replicate the *entirety* of human(-level) intelligence. Human intelligence concerns both *what* people can exhibit in terms of intelligent behavior and *how* they achieve this. Human-level intelligence likewise strives for the former, but generalizes the latter to any computational means of achieving the former. Such work may sit firmly within artificial intelligence or cognitive science, or it may span both. In the *pure* cases, the desiderata generally align well with the encompassing field, with the notable added complexity of how the focus on *entirety*, and thus on an integrated whole, skews things. For example, locally optimizing each capability – in terms of either raw performance or match to human data – to compete with the best techniques in isolation may not be the best approach to developing the most globally optimal system. In the *mixed* case, the two sets of desiderata must be blended in some way. A simple union of the two fields' desiderata would most facilitate publication, by enabling easy alignment with existing standards and norms, but this is typically not feasible in practice.

In the remainder of this section we explicate the four desiderata briefly mentioned in Section 1, which are hypothesized to be appropriate for work that involves both of these complexities; that is, models of the entirety of human(-level) intelligence that span both artificial intelligence and cognitive science. Some of these desiderata are variations on long-standing ones in cognitive architectures/systems, while others make additional assumptions about what is desired. Either way, by delving down below the general notion of "understanding and/or replicating the entirety of human(-level) intelligence" they can help to both guide and evaluate the research. This section concludes by mapping the desiderata onto the unifying notion of a *cognitive hourglass*.

2.1 Grand Unification

The essence of any cognitive architecture is the combination of capabilities hypothesized to underlie thought and intelligent behavior. A traditional *unified* cognitive architecture attempts to integrate the cognitive capabilities required for human(-level) intelligent behavior, with appropriate sharing of knowledge, skills and uncertainty among them. A *grand unified* architecture goes beyond this, in analogy to a grand unified theory in physics, to attempt to include the crucial pieces missing from a purely cognitive theory. The goal can be thought of as being to cover all of the time scales in Newell's analysis of human cognition – Table 1 – which decomposes human performance into broad social, rational, cognitive and biological bands, with each band further decomposed into a trio of layers that



Band	Scale (sec)	Time Units	System
Social	10^7	months	
	10^6	weeks	
	10^5	days	
Rational	10^4	hours	Task
	10^3	10 min	Task
	10^2	minutes	Task
Cognitive	10^1	10 sec	Unit task
	10^0	1 sec	Operations
	10^{-1}	100 ms	Deliberate act
Biological	10^{-2}	10 ms	Neural circuit
	10^{-3}	1 ms	Neuron
	10^{-4}	100 µs	Organelle

Table 1: Time scales in human cognition (adapted from Newell, 1990).

individually span a single order of magnitude (Newell, 1990). This analysis includes not only the full range of cognitive behavior but also the relevant subcognitive behavior, in the biological band. Anderson (2002), for example, discussed the specific possibility of a single model spanning seven of these layers. In Sigma, we are striving to cover all four bands in the long term, but as described further in Section 2.2, with respect to the biological band this ambition extends only as far as a graphical abstraction of it.

Another way to think about grand unification is in terms of unifying across the full arc from perception and attention, up through cognition, and back down to intention and action. This shifts issues of embodiment, grounding and interaction into the foreground, to converge with work on (intelligent) robot and virtual human architectures (Coste-Manière and Simmons, 2000; Murphy, 2000; Badler, 1997; Hartholt et al., 2013), but without then relegating traditional cognitive concerns to the background.

So grand unification involves spanning both cognitive and subcognitive functionalities as well as both central thought processes and more peripheral perceptual and motor processing. It also involves spanning both the traditionally studied rational thought processes and what may seem like the less rational processes involved in emotion and affect, but which capture what can be considered the *wisdom of evolution* in natural systems by providing non-voluntary pressures that have proven – via natural selection – to be adaptive over the long-term.

Although creating a (grand) unified cognitive system was the originating grand challenge of AI, it is now more a definition of the span of topics covered by the field, and even then, many of the topics have branched off into their own separate disciplines, with only residuals left within the main line of AI itself. Similarly, (grand) unification is more of a way of defining the span of topics within cognitive science than an explicit goal for most researchers in that interdisciplinary field. (Grand) unification is, however, an important goal in Artificial General Intelligence (AGI), which has explicitly resurrected the original AI grand challenge, and in various subfields that focus explicitly on cognitive systems and architectures, including work on architectures for intelligent robots, agents and virtual humans.

2.2 Generic Cognition

Generic cognition concerns a deliberate conflation of natural and artificial cognition. Ideas and results from both may be freely intermingled in guiding and constraining the development of an artificial mind that may serve both as a useful artifact on its own and as a model of natural minds at some level of abstraction. Unlike grand unification, generic cognition is not about combining multiple capabilities per se, but about finding the essential underlying commonality among their natural and artificial variants. The recent notion of *substrate independence* – that mental states can exist on a variety of different organic and inorganic substrates (Bostrom, 2001) – captures one aspect of what is meant here by "some level of abstraction." But the general idea is much older, going back at least to Marr's (1982) three levels of analysis of information processing systems and Newell's analysis of time scales in cognition (Table 1). The work on Sigma – as with Soar before it – has been heavily influenced by Newell's analysis.

At a high level, the key questions Newell's analysis raises for cognitive modeling are, what are the distinctive properties of the different bands, and of the time scales within each band? within which bands and time scales do different human functionalities fall? and, can a cognitive system provide comparable layers and a comparable distribution of functionalities across the layers? A precise mapping is difficult between Sigma and this analysis, but Sigma's graphical and cognitive architectures are intended to correspond roughly to portions of the biological and cognitive bands, respectively. Newell's and others' work with Soar helped establish the latter part of this mapping. The known mapping between graphical models and neural networks (Jordan and Sejnowski, 2001) bolsters the former part of it, but with the graphical architecture clearly abstracting away many of the lower-level details of the biological band. From the perspective of generic cognition, it may even be worth asking whether it would be better to replace Newell's "natural" notion of a biological band with the more generic notion of a *graphical band* that subsumes neural networks, graphical models, and other highly parallel graphical/network models.

Although not always stated explicitly in writings on Soar, the desideratum of generic cognition is a direct inheritance from the work practice in that project. In Sigma, the emphasis on functionality, the appeal to graphical models, and the prospect of applications in virtual humans and intelligent agents/robots are clearly most tied to the artificial side. The relevance of existing cognitive architectures, the concern with time scales in cognition – for example, distinguishing what happens within a single *deliberate act* at the bottom of the cognitive band (~100 msec) versus a sequence of deliberate acts at longer time scales – and the prospect of (unified) theories of human cognition are clearly most tied to the natural side. The initial emphasis in modeling within Sigma is on getting the gross structures right, with the expectations that more detailed matches will follow (typically starting with low hanging fruit).

Cognitive science might have developed as the science of generic cognition, but it has instead mostly followed a narrower path, leveraging insights from both natural and artificial systems in modeling natural cognition, that has led it to bypass how these insights could also help develop artificial (general) intelligence. Two relatively new communities, Biologically Inspired Cognitive Architectures (BICA) and Advances in Cognitive Systems (ACS), are more committed to generic cognition, but with the former emphasizing the low-level subcognitive aspects and the latter the high-level cognitive aspects.

2.3 Functional Elegance

Functional elegance implies combining broad functionality – generically cognitive grand unification in this case – with simplicity and theoretical elegance. The goal is something like a set of *cognitive Newton's laws* that yield the required diversity of behavior from interactions among a small set of general primitives. Illustrative analogies can also be drawn to chemistry, where a diversity of molecules arises from interactions among a relatively small set of elements; and to mathematics, where large numbers of theorems can be proven from small sets of axioms. Developing new capabilities in such a framework typically amounts to more of a discovery process than simply an implementation process, with many of the functionalities exhibited within Sigma most appropriately being thought of as being *deconstructed* (in terms of a combination of more primitive capabilities) or *proven* (in an informal sense from ground mechanisms). As a result, many of the results are demonstrations that an appropriate deconstruction/proof can be found with little to no extensions at lower levels. Functional elegance bears a relationship to the notion of *emergence* (O'Connor and Wong, 2015), but is a weaker notion in not necessarily implying that nothing need explicitly be added as we move upwards.

Whatever approach is taken to achieving functional elegance, if the primitives are combinable in a flexible enough manner, new capabilities continue to flower without the need for new architectural modules, and integration occurs naturally through shared primitives. There is also increased potential for establishing theoretical claims about such a model, and for yielding systems that are ultimately easier to understand, use and maintain. A functionally elegant theory may also provide a deeper and fundamentally better *explanation with reach* (Deutsch, 2011).

Still, functional elegance is far from generally accepted as a goal for cognitive systems, due presumably to doubts that such a theory is within reach or to an assessment that a diverse core provides a better path to developing full cognitive systems. When modeling human behavior is of concern, the importance of functional elegance is even more debatable than when implementing artificial cognition. Against it is the argument that evolution has been tinkering with natural minds, and extending them in ad hoc ways, for millions of years (Minsky, 1986). But in its favor is the argument from rational analysis: that human minds have developed to provide (computationally bounded) optimal responses to their environments (Anderson, 1990; Chater and Oaksford, 1999). Moreover, if two equally accurate models of human cognition were to exist, one of which was functionally elegant and the other not, Occam's razor would favor the former.

2.4 Sufficient Efficiency

Sufficient efficiency implies implementing cognitive systems in a manner that is efficient enough to support their anticipated uses. It takes a *satisficing* perspective on the classic computer science and AI goal of optimization, in the spirit of Herb Simon's proposal that humans are satisficers rather than optimizers in making decisions (Simon, 1956), and as partial protection from overdoing local optimization. Still this does not mean that timing is unimportant; it is in fact still crucial. What it does imply, though, is that the criterion of comparing timings across systems – who beats whom – is to be replaced by how well the timings match the needs of the models and applications that are relevant to human(-level) intelligent behavior. Comparisons across systems are still relevant, but only in terms of which systems yield more appropriate timings.

Sufficient efficiency has both a real-time component and a model-time component. The real-time component concerns the speed, and at times the boundedness, of execution. This has a theoretical aspect, concerning whether an implementation is even possible that is computable and either sufficiently tractable or boundable; for example, whether the core *cognitive cycle* – which is to map onto ~50 msec in humans (Card, Moran and Newell, 1983; Anderson, 2007; Kieras and Meyer, 1997; Laird, 2012; Rosenbloom, 2012c), or roughly half of a *deliberate act* from Table 1 – is provably bounded, or at least tractable. But the larger part of this is pragmatic, ensuring that the system executes quickly enough on available computers for anticipated applications, whether controlling robots in the physical world or providing virtual humans that can interact with people in real time. For cognitive modeling, it is sometimes possible to get by with just simulated real-time. However, even there speed becomes increasingly critical as experiments scale up, such as with very large memories (Douglass et al., 2009; Derbinsky et al., 2010). Efficiency has been a continuing concern in the development of Sigma, starting with an initial appeal to the breadth of state-of-the-art performance yielded by graphical models, and extending to further optimizations that are discussed partly in Section 5.2.4 and in further detail in Rosenbloom (2012c) and Rosenbloom, Demski and Ustun (2015). Yet, significant additional work is still necessary.

The model-time component of sufficient efficiency concerns whether activities execute at appropriate human time scales when counting the number of primitive model steps, independent of how much real time is required to execute the model steps. In Soar, for example, the model steps are decision cycles, each of which is to correspond to ~50 msec of human time. If humans can execute some aspect of processing in less than 50 msec, then implementing it via multiple decision cycles would be insufficiently efficient in model-time. In principle, sufficient efficiency – whether model-time or real-time – can lead to tradeoffs with functional elegance when the most elegant way to implement a capability is not sufficiently efficient. Such situations are worth particular attention as critical challenges to the *feasibility* criterion introduced earlier.

2.5 Cognitive Hourglass

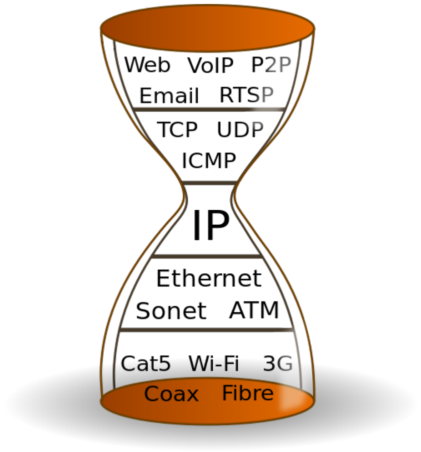


Figure 2: Internet Hourglass.

Physics analogies were used in explaining the first desideratum, but there is a different analogy, from computer networking, that can provide further insight into all four desiderata and how they relate to each other (Rosenbloom, 2011a). The *Internet Hourglass* (Deering, 1998) – Figure 2[1] – ties together the diversity of network applications and implementation technologies via the narrow waist of the Internet Protocol (IP). In the network protocol stack, IP divides the world into "everything on IP" (i.e., the diversity of applications implementable on top of IP) versus "IP on everything" (i.e., the diversity of implementation technologies available for IP). In the corresponding cognitive hourglass, shown in Figure 3, the move from unification to grand unification amounts to a horizontal expansion of the top of the hourglass. Generic cognition thickens the hourglass, making it 3D, so that its entirety covers both natural and artificial cognition. Functional elegance corresponds to the upper half of the hourglass, where behavioral diversity at the top is grounded in the small elegant core provided by the waist. Sufficient efficiency spans the height of the hourglass, with the overlap in the top half reflecting the tradeoffs that may occur between it and functional elegance.

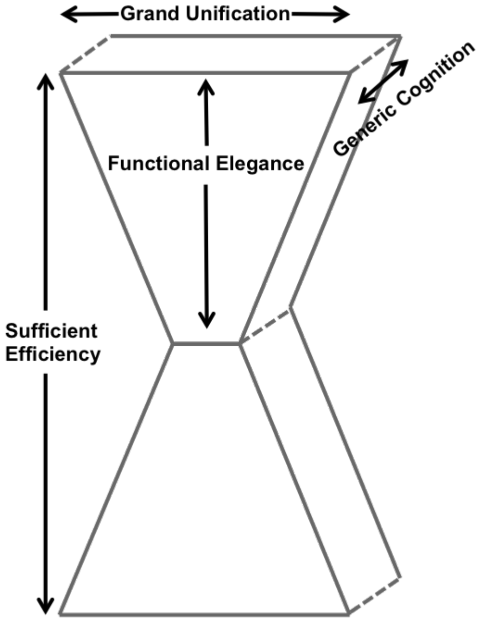


Figure 3: Cognitive Hourglass.

The four layers of Sigma shown in Figure 1 can be mapped onto this cognitive hourglass, although it has taken a few years to understand how to do this properly. In the early work on Sigma, when its cognitive architecture and cognitive idioms (i.e., its analogues of programming idioms) were combined conceptually into a graphical cognitive architecture, the bottom two layers – to be more precise, the graphical architecture and its implementation in Lisp – were also combined, into what was called an *implementation layer*, which was so named because it implemented the cognitive architecture (Rosenbloom, 2011a). At that time, this implementation layer was mapped onto the waist of Sigma's cognitive hourglass. However, two significant adjustments have since been made to this mapping to yield the hourglass in Figure 4. First, recent work on optimizing Sigma's implementation code (Rosenbloom, 2012c; Rosenbloom, Demski and Ustun, 2015) has made it

[1] Reproduced under the Creative Commons Attribution-Share Alike 3.0 Unported, 2.5 Generic, 2.0 Generic and 1.0 Generic license.

obvious that the layers do widen in moving down from the graphical architecture to the code that implements it, via alternative special-case implementations and optimizations, and that thus only the former should be at the waist, with the latter below it.[2] The implementation – now of the graphical architecture rather than the cognitive architecture – is still shown in the figure, but in a distinctive manner to signify that it isn't an identifiable layer in Figure 1. Second, separating cognitive idioms from the cognitive architecture revealed how truly minimal the latter was when considered on its own, although still not quite as minimal as the graphical architecture. Both the idioms and architecture are shown in Figure 4, with the same caveat about the former as about the implementation.

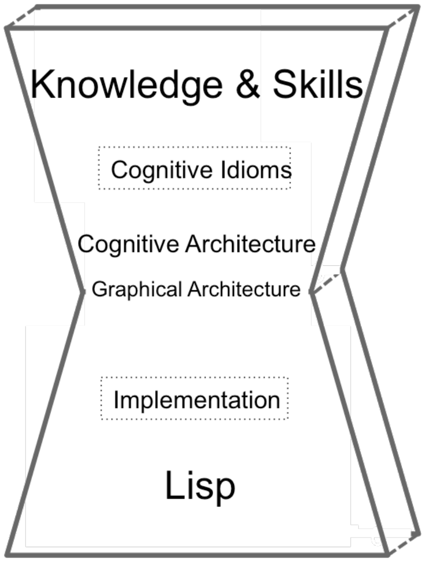


Figure 4: Sigma Hourglass.

3. Why Sigma?

Cognitive architectures are hypotheses about the fixed structures, and their interactions, underlying intelligent behavior in natural and/or artificial systems. In simpler words, they are hypotheses about the nature of mind. When implemented, they are typically large, complex software systems. Given the range of cognitive architectures that exist (see, for example, Langley, Laird and Rogers, 2009; Goertzel, 2014), and the years – if not decades – required to fully develop a new one, a compelling rationale is necessary for it to be worth developing a new one. With Sigma, the impetus stemmed from the demands imposed by the four desiderata – which no existing architecture fully satisfies – plus the revolutionary potential of graphical models to do so.

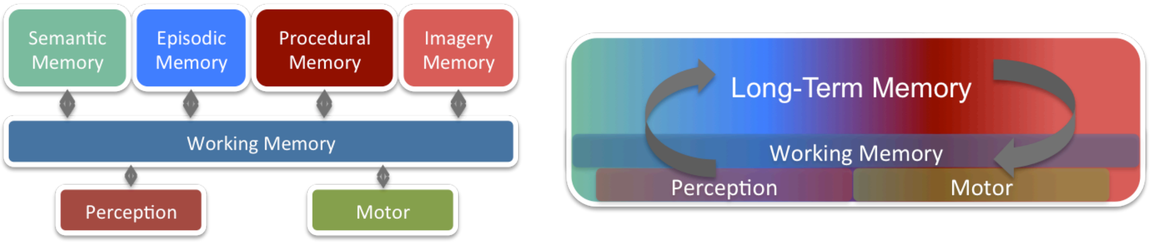
The two foundational systems in cognitive architecture are ACT-R and Soar, both of which have been under development in some form for over three decades. ACT-R and its lineal ancestors, such as ACT* (Anderson, 1983), have appeared in many variations over the years in their drive to model a broad swath of the microstructure of human cognitive behavior. Although ACT-R is primarily focused on modeling human cognition, and largely symbolic in nature, its models have been used in applications, such as human performance modeling in military simulation (Best et al., 2002), and it has always had a central, subsymbolic, activation component. It has also more recently been mapped onto regions of the brain (Anderson, 2007) and hybridized with neural networks for visual perception (Jilk et al., 2008). Two of the lessons from the history of ACT-R that go beyond the standard model and have influenced Sigma are (1) the pervasive importance of subsymbolic processing even in the context of traditional symbolic processing, and (2) how to successfully use an architecture in modeling the microdetail of human cognition.

Soar began as a symbolic cognitive architecture geared towards building artificially intelligent systems, but with a backstory that included building a unified theory of human cognition (Newell, 1990). In its early days it was based on a working memory of object-attribute-value triples and a long-term memory based on a parallel rule system (Laird, et al., 1987). Firing rules in long-term memory could change working memory and yield preferences for operators to be applied (by more rules) in problem spaces (Newell et al., 1991). This behavior was driven by

[2] The role of the graphical architecture at the waist of this hourglass bears a strong resemblance to the notion in Alchemy of Markov logic providing an *interface layer* for AI systems (Domingos and Lowd, 2009).

a decision cycle consisting of two phases: elaboration, during which rules fired; and decision, during which operators were selected. When decisions were problematic, impasses occurred that led to reflection. Soar's learning mechanism (chunking) created new rules that captured the reflective problem solving that resolved impasses. Since these early beginnings, additional long-term memories have been added to Soar – for semantic, episodic and imaginal knowledge – and several new learning mechanisms have also been added, including semantic, episodic and reinforcement (Laird, 2012). Some aspects of Soar have also recently taken on a subsymbolic flavor, such as numeric preferences (for decision making and reinforcement learning) and activation (for attention). A number of these changes have been influenced strongly by ACT-R.

Sigma can be viewed as an attempt by one of the original co-developers of Soar to retain what still seems right about it after all of these years – at least to him – while radically rethinking those core assumptions that appear to be holding it back. The additional lessons from the history of Soar that have influenced Sigma include (1) the use of problem spaces to structure cognitive behavior; (2) the importance of the two-phase structure of the cognitive cycle; (3) the functional elegance of a nested three-layer model of control that layers reactivity within deliberation and deliberation within reflection; (4) the importance of an automatization/knowledge-compilation mechanism such as chunking for real-time behavior in the presence of complex reflective reasoning; (5) the push for uniformity that existed during Soar's early days, with Allen Newell's advice of "listening to the architecture" rather than immediately adding new modules when extending Soar's functionality; (6) how to approach generic cognition by balancing the construction of functional AI systems with modeling of human cognition; and (7) the importance and difficulty of integrating capabilities implemented via knowledge on top of the architecture.



(a) Modular architecture

(b) Functionally elegant architecture

Figure 5: Caricature of modular versus functionally elegant architectures.

Rather than implementing a new architecture from scratch, could we instead have directly built upon either ACT-R or Soar? The simple answer is *no*. Both architectures do achieve significant levels of generic cognition, although ACT-R is stronger on the natural side and Soar is stronger on the artificial side. Soar, and possibly ACT-R, is also strong on sufficient efficiency. However, both are built from a diverse set of modules that are implemented directly in some ground programming language, and require more modules of quite different sorts to expand from unification to grand unification. Soar has, for example, been connected to external speech, vision and SLAM (simultaneous localization and mapping) modules at various times. Figure 5 caricatures the difference between a diverse/modular architecture and a functionally elegant one. Figure 5(a) is roughly based on the latest versions of Soar, but the general approach applies to other diverse architectures such as ACT-R, CogPrime (Goertzel et al., 2014) and PolyScheme (Cassimatis, 2002) as well. Figure 5(b) maps roughly onto Sigma's functionally elegant approach, with different functionalities arising as distinct specializations and combinations across a common underlying base.

To yield 5(b) from 5(a) would require reimplementing the modules making up the latter in a more functionally elegant manner; that is, exactly what we are trying to do with Sigma by starting out with a *hybrid* (discrete + continuous) *mixed* (symbolic + probabilistic) approach from the very beginning, via a grounding in graphical models that span signals (continuous), probabilities (discrete and continuous) and symbols (discrete).

One of the major roadblocks Soar ran into over the years was that, although its original small set of architectural mechanisms could be shown to yield a wide diversity of intelligent behaviors, due to control conflicts, too often these derived behaviors were in turn too difficult to reuse when constructing more complex behaviors and systems. This is what led, for example, to the later inclusion of a semantic memory module, even though the basic capability was demonstrated years before via Soar's existing rules and chunking mechanism (Rosenbloom, et al., 1987). In contrast, a recent analysis explains how Sigma can implement semantic memory and learning in a functionally elegant manner, as a *supraarchitectural cognitive idiom* that can integrate straightforwardly with other such capabilities (Rosenbloom, 2015). Section 6 includes numerous examples of how Sigma has enabled the specification and reuse of a diversity of intelligent capabilities, while Section 7 demonstrates integrations across significant numbers of them.

When leveraging functional elegance in comparing Sigma with existing architectures like Soar and ACT-R, an analogy with the Copernican revolution can be illustrative. Copernicus revised a fundamental assumption at the heart of our understanding of the solar system – shifting from a geocentric to a heliocentric view – to yield a new model that was simpler and more elegant than the prior Ptolemaic model, although it didn't necessarily match the astronomical data any better, and in fact appears to have been slightly worse. The Copernican model also had some additional rough edges itself, retaining some of the smaller epicycles from the Ptolemaic model while dispensing with the larger ones. Yet by shifting to a simpler and ultimately more correct core assumption, it started down a path that enabled Kepler to refine it further by generalizing the orbits from circular to elliptical. In a similar manner, with Sigma we often first yield a Copernican result that reformulates an existing capability to be more elegant, even though it may still have rough edges of its own and may not be an improvement in functionality or modeling. In some cases, though, we have also been able to take the next step of refining this to a more Keplerian result.

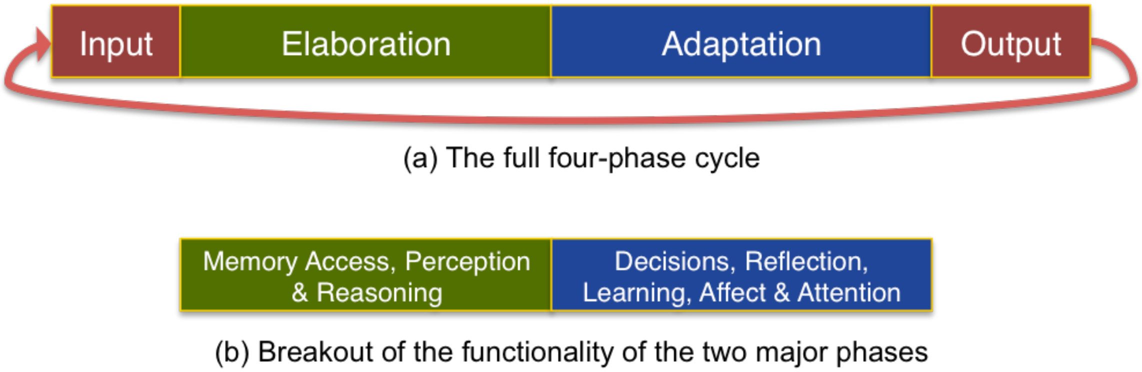
What about starting with architectures other than Soar and ACT-R? Most are constructed from a diverse set of modules, like ACT-R or Soar, including some like CogPrime and PolyScheme that take strong positions in favor of this form of diversity. Starting with any of these architectures would be as difficult, if not more difficult, than starting with ACT-R or Soar. The biggest exception is AIXI, a proposal for *universal artificial intelligence* (Hutter, 2005) that strives for an extreme version of functional elegance, in terms of a single equation that covers all of intelligence. It also strives for a form of grand unification in its appeal to universality. However, it is completely focused on artificial intelligence, and has major issues with sufficient efficiency; as originally defined, it wasn't even computable, although computable approximations are being explored (Veness et al., 2011). A bigger issue with respect to the goals of Sigma is that it does not support exploration of the graphical architecture hypothesis, incorporating neither the lessons from cognitive architectures nor those from graphical models. Fundamentally, building a cognitive architecture based on graphical models at this point requires starting from scratch.

It would be conceivable to start from a different, and possibly preexisting, graphical architecture or language, rather than the homebrewed version used in Sigma. There are many more candidates available now than existed when the work on Sigma was begun, but that is a different question, and one that still would require replication of all of the kinds of work described in this article above the graphical architecture.

4. Sigma Cognitive Architecture

In this section the Sigma cognitive architecture will be presented without reference to the graphical architecture that supports it at the level below. Early descriptions of Sigma began with the graphical architecture because, given the state of understanding at that time, it was unclear how to explain and justify the cognitive architecture without first covering the graphical architecture, whose beginnings predated the cognitive architecture and whose capabilities inspired its design. It is now, however, possible to explain the cognitive architecture on its own, hopefully leading to a clearer and more autonomous understanding of it. Presentation of the graphical architecture and the way the cognitive architecture maps onto it have therefore been deferred to Section 5. Furthermore, the conceptual structure of the cognitive architecture is presented first, without direct reference to the cognitive language that is induced by the architecture (Section 4.4).

At the core of the cognitive architecture is a *cognitive cycle* (Figure 6) that is intended to correspond to the ~50 msec cycle in humans, and which thus also maps approximately onto the decision cycle found in Soar (Rosenbloom, 2011c) and onto the corresponding cycles in other architectures. Sigma's cycle has two minor and two major phases. Input and output, at the beginning and end of the cycle, are minor phases intended to perform just the basic transduction that enables the system to interface with its environment. Both grand unification and functional elegance direct that perceptual and motor processing should as much as possible happen inside the system, in a uniform manner with other forms of processing: that is, via processing within the two major phases.


Input
Elaboration
Adaptation
Output
Memory Access, Perception & Reasoning
Decisions, Reflection, Learning, Affect & Attention

(a) The full four-phase cycle

(b) Breakout of the functionality of the two major phases

Figure 6: Sigma's cognitive cycle.

In the first major phase (Section 4.1) knowledge about the current situation is processed, including retrieving it as necessary and drawing conclusions from it. It is named the *elaboration phase* because its role is, much like that of the elaboration phase in Soar, to elaborate in a largely monotonic manner (i.e., in a non-decreasing manner) what is understood about the current situation. However, the kinds of knowledge that can be represented and the kinds of processing that can occur have both been greatly expanded, spanning hybrid and mixed representations and subsuming symbol, probability and signal processing. Sigma's elaboration phase thus involves both symbolic and sub-symbolic "knowledge," and performs not only normal cognitive reasoning but also perceptual processing and other critical forms of sub-cognitive processing. Functionally, the elaboration phase updates distributions over variables, in parallel (at least logically) and repeatedly, until *quiescence* is reached: that is, until no more updates are available. It does not, however, either make choices or learn.

The second major phase (Section 4.2) is largely non-monotonic, making choices and persistent changes to memories based on the quiescent distributions engendered by the elaboration phase, while also engaging in meta-architectural processing that senses and modifies how the architecture itself works. This *adaptation phase* roughly maps onto the decision phase in Soar, but the new name explicitly acknowledges that a wider range of adaptation is occurring that goes beyond just the selection of operators to apply to states. Adaptation here spans short-term modifications to working memory – including but not limited to selection of operators and the reflection that can result when decisions about operators are not possible – as well as alterations to the system's affective and attentional states. It also includes long-term modifications to long-term memory; that is, learning.

Consider a simple *classification scenario* that spans all four phases of this cycle, involving perception, classification, selection and action:

Input: A set of features is perceived [`color=silver`] for an object `o1`
Elaboration: A classifier (Figure 7) yields a distribution over the implied concept for `o1`; and then a rule proposes an operator that prints the concept
Adaptation: An operator is selected for printing the concept [`walker`]
Output: The concept name is printed [`Concept: walker`]

This scenario will serve as a *running example* through this section and the next few. It is far from comprehensive enough to cover everything that will be seen, but it does encapsulate enough of the core ideas to be illustrative – including procedural (rule) and declarative (semantic) memory, operator selection, perception and action – and it can be extended with semantic and episodic learning. To cover where it falls short, a variety of other implemented examples will also be introduced as necessary, including several – such as the Eight Puzzle and a small, discrete 1D world – to which we will refer back a number of times.

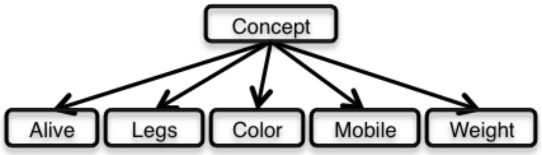


Figure 7: Simple naïve Bayes classifier (as a Bayesian network).

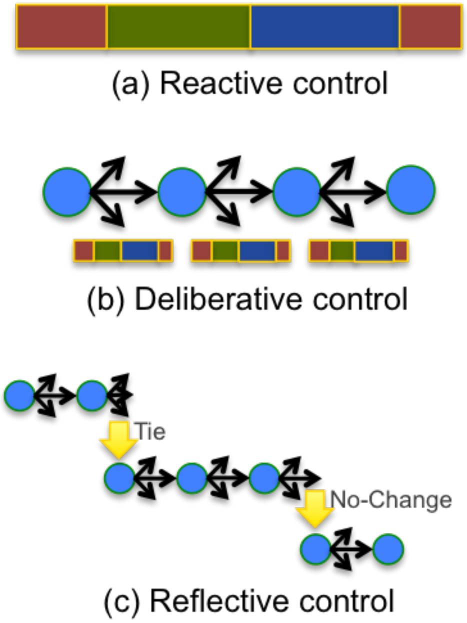


Figure 8: Tri-level control structure.

As in Soar, Sigma's cognitive cycle is central in supporting the *problem space computational model* (Newell et al., 1991), where goals are achieved by searching in problem spaces that are specified in terms of sets of states and operators. Also as in Soar the cognitive cycle is the basis upon which a functionally elegant *tri-level control structure* is constructed in a nested manner: (1) a *reactive* capability that is based on a single cognitive cycle (Figure 8(a)) serves as the inner loop of (2) a *deliberative* capability that is based on a sequence of cognitive cycles (Figure 8(b)), and which itself serves as the inner loop of (3) a *reflective* capability that is based on impasses in decision making plus processing at the metalevel (Figure 8(c)). Section 4.3 further discusses this nested control structure.

4.1 Elaboration Phase

The elaboration phase is largely about *memory*, and about spinning out the consequences of what is in memory. Sigma's cognitive memory is conceptually partitioned into a *long-term memory* (LTM), a *working memory* (WM) and a *perceptual buffer* (PB) (Figure 9). Like the early versions of Soar, Sigma has only one long-term memory and one working memory. However, in Soar, perception is stored directly in WM, whereas in Sigma it is most cleanly considered as occupying a separate memory that maintains the contents of perception until it is changed, while also passing the contents on to WM for consideration.

In effect, the elaboration phase combines the knowledge that is available within these three memories at the beginning of the phase to yield new WM content at the end of the phase. Output in Sigma is still based directly on the contents of WM, as in Soar, with no distinct output buffer defined, but this may change if further experience with motor control should indicate a need for it.

Distinctions between different types of LTM, WM (and PB as appropriate) – such as procedural, declarative (whether semantic or episodic), imagery, visual and auditory – occur above Sigma's cognitive architecture, via cognitive idioms (Section 6). In other words, these varieties of memories are deconstructed in terms of the more basic memory structures provided by the cognitive architecture. One consequence of this is that portions of perceptual, working and long-term memory may be ascribed as necessary to each idiomatic memory that is defined.

All three memories contain *functions*, each of which specifies real values over a domain defined by a set of variables. Long-term memory also contains *structures* that are defined as patterns over multiple functions. The remainder of this subsection explores these two topics.

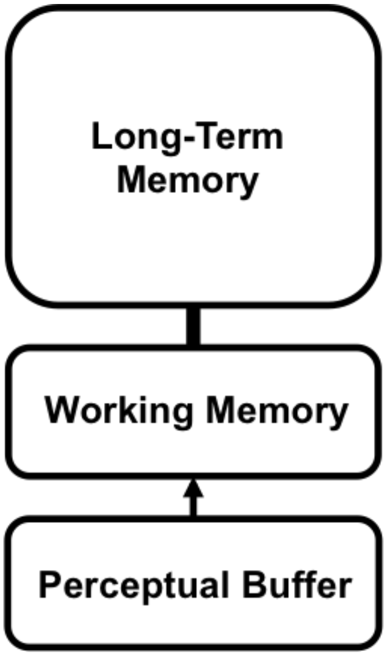


Figure 9: Cognitive memories.

4.1.1 FUNCTIONS

Functions yield real values over domains defined via zero or more variables. Each variable within a function can be numeric – either continuous or discrete – or symbolic, and mixtures of variable types across dimensions of individual functions are possible, and in fact common. The real values in the range of the functions are typically non-negative, due to constraints imposed by the graphical models at the level below, but at least one important situation in which negative values matter, and are in fact allowed, will be discussed shortly. Functions may be normalized over one or more variables – for example, to yield probability distributions – but they need not be.

Consider a few examples. One of the simplest represents the conditional probability of the color of an object given its category – **P**(*Color* | *Category*) – as a function over two symbolic variables that is normalized along the *Color* variable. For each combination of *Color* and

Category, the value of the function is the probability of that *Color* given that *Category*. A more complex example of a similar sort is the mental representation of a map during Simultaneous Localization and Mapping (SLAM) (Bailey and Durrant-Whyte, 2006). Figure 10 shows the small, discrete, 1D world in which the early experiments with SLAM in Sigma were done. In a more realistic 2D version, there are three variables: *X* and *Y* for the two spatial dimensions, and *Object* for the objects that may exist in the world. The spatial dimensions could in principle be continuous, but the implementations of SLAM in Sigma have so far used discrete numeric dimensions/variables. *Object* is a symbolic variable. What is represented and in fact learned here is a conditional probability distribution corresponding to **P**(*Object* | *X*, *Y*), with normalization along the *Object* variable.

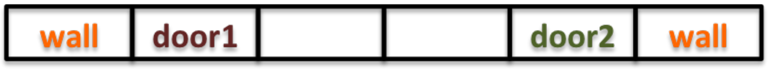


Figure 10: Small 1D world bounded by indistinguishable walls and with two distinct doors.

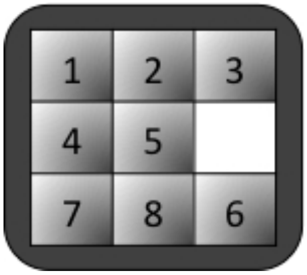


Figure 11: Eight Puzzle board.

The board in the Eight Puzzle (Figure 11), a classic sliding-tile puzzle, can be represented similarly, although the current implementation in Sigma uses two continuous, rather than discrete, spatial variables plus a discrete numeric variable for the tile. The resulting *hybrid* function, when normalized along the *Tile* represents **P**(*Tile* | *X*, *Y*), as shown in Figure 12. Effectively, a distinct 2D plane is defined for each tile, with the value of 1 over the entire region in the plane covered by the tile and 0 everywhere else. This representation can thus be viewed as both an *occupancy grid* for the tiles and a *mental image* for the board.

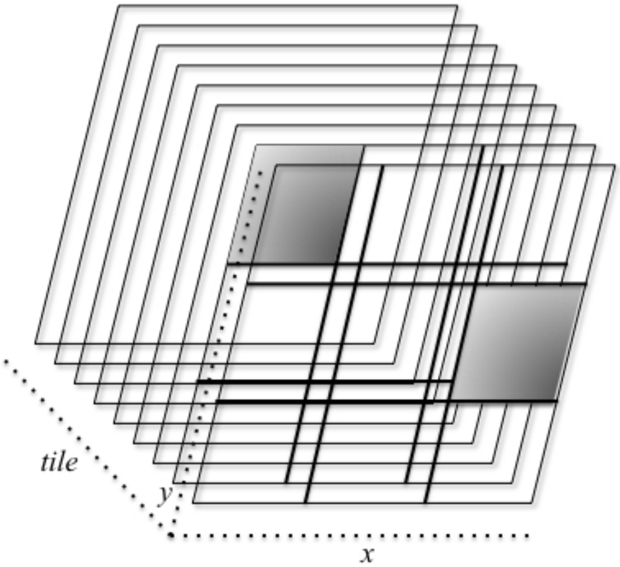
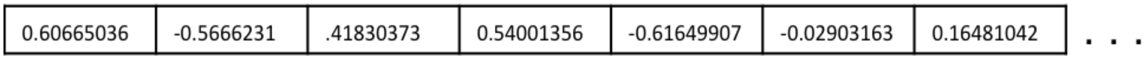


Figure 12: Partial visualization of a hybrid representation for the Eight Puzzle board, with one bounded (continuous) 2D plane per tile (including the blank). Only the regions corresponding to the blank and the first tile are highlighted (in gray).

We will finish up these examples of cognitive functions with two simple but rather different forms. The first example encodes whether one object is above another via two symbolic variables for objects, with a functional value of 1 when the first object is above the second object, and a functional value of 0 elsewise. The result is an unnormalized Boolean function – **Above**(*Object1*, *Object2*) – that serves as a symbolic relation between the variables. The second example encodes *distributed vectors*, like in Figure 13 and as used in neural networks, deep learning, word embeddings and holographic models (Rumelhart et al., 1986; Bengio et al., 2003; Collobert and Weston, 2008; Mikolov et al., 2013; Mnih and Kavukcuoglu, 2013; Turney and Pantel, 2010; Jones and Mewhort, 2007). Such vectors are typically used, for example, to learn and represent the meaning of words or concepts in a form that enables vector similarity – for example, as computed by the standard cosine similarity metric – to correlate with meaning similarity. We have initial experience in learning and using such vectors in Sigma – under the name DVRS (Distributed Vector Representation in Sigma) – for example, in supporting analogical reasoning among words (Ustun et al., 2014). Distributed vectors do require negative functional values, as shown in the figure, so as to exploit all four quadrants of the vector space. Normalization must then convert such functions into unit vectors rather than ensuring that they sum to 1 – that is, it must yield vector rather than probabilistic normalization.



0.60665036	-0.5666231	.41830373	0.54001356	-0.61649907	-0.02903163	0.16481042	. . .

Figure 13: Example continuous vector for representing word meaning in a distributed fashion.

When Sigma is used to represent multiple agents within a single software image, all of the functions that are to be agent specific are augmented with an additional discrete variable – either symbolic or numeric – for the *Agent*. For example, in SLAM the function is extended from **P**(*Object* | *X*, *Y*) to **P**(*Object* | *Agent*, *X*, *Y*). This enables multiple agents to run independently within one instance of the cognitive architecture, rather than requiring a new instance for each.

Both working memory and the perceptual buffer are composed entirely of functions. The values of perceptual functions are determined directly by what appears in the input phase of the cycle. This is combined during the elaboration phase with the functions retained in WM from the previous cognitive cycle and what is retrieved from long-term memory to yield new functions in WM. This may then cue further retrievals from LTM until quiescence is reached. LTM also contains functions, but structures also play a significant role.

4.1.2 STRUCTURES

In addition to functions, long-term memory also embodies more complex structures that are defined as patterns over multiple functions. As a result, a more involved retrieval process is necessary for LTM that ends up being inextricably bound together with inference: the structures themselves implicitly define how both are to proceed.

One form LTM structures can take in Sigma is as fragments of probabilistic networks. For example, combining a prior distribution over a category with conditional distributions over various features given the category can yield a *naïve Bayes classifier* – Figure 7 – as found in generative concept learning. Likewise, combining a probabilistic transition function that relates the previous state in a sequence to the current state in the sequence – **P**(S_i | $S_{i\text{-}1}$) – with a distribution over a perceived feature given the state – **P**(F_i | S_i) – can yield a *hidden Markov model* (*HMM*), as shown in Figure 14 and used in speech processing. Or, combining a pair of probabilistic transition functions, one each for a pair of agents, A and B, can yield a POMDP for solving the *Ultimatum Game*, as shown in Figure 15, where A starts with a fixed amount of money (3 in this example), and offers a portion of it to B, who then accepts or rejects the offer. If B accepts, it receives the offered amount, while A keeps the remainder. However, if B rejects the offered amount, both get 0 (Pynadath et al., 2013).

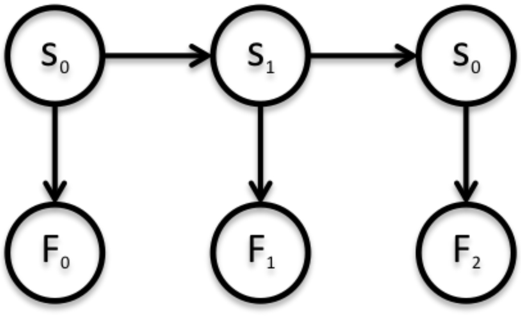


Figure 14: Three-stage hidden Markov model (as a dynamic Bayesian network).

Sigma's LTM structures can also take the form of symbolic rules, built from conditions and actions over symbolic relations (Figure 16). As with Soar, such rules may play many roles in Sigma, from providing basic inferences (Figure 16(a)), to encoding plans and generating control information (Figure 16(b)), to specifying how to apply selected operators.

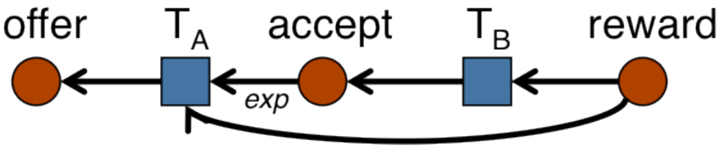


Figure 15: Trellis for the Ultimatum Game (as a factor graph).

Beyond directly enabling the encoding of probabilistic networks and rules, Sigma's LTM can also support various micro- and macro-level combinations of them. Micro-level combinations combine aspects of both probabilistic networks and rules into mixed and hybrid forms. They may be as simple as probabilistic rules, or as complex as some we shall discuss later. Macro-level combinations combine multiple distinct structures. This could simply involve chaining across rules, or combining multiple distinct probabilistic networks, such as combining a conditional random field for object perception, a probabilistic localization network (a key part of SLAM), and a POMDP for action choice, into a larger composite network that goes from perception through localization to decision making (Figure 17). Or it could involve combining rules and probabilistic networks, as when a rule is used to select either the most probable concept in the classification scenario or the single best word from a hidden Markov model for speech recognition (Joshi et al. 2014).

$\text{Above}(a, b) \wedge \text{Above}(b, c) \rightarrow \text{Above}(a, c)$

(a) Transitive rule for *above*.

$\text{Concept}(\text{o1}, c) \rightarrow \text{Operator}(c)$

(b) Rule for selecting the concept for object o1 as the operator in the classification scenario.

Figure 16: Two example rules. Variables are *italicized* while constants are not.

The details behind all of this will become clearer as the cognitive language is introduced in

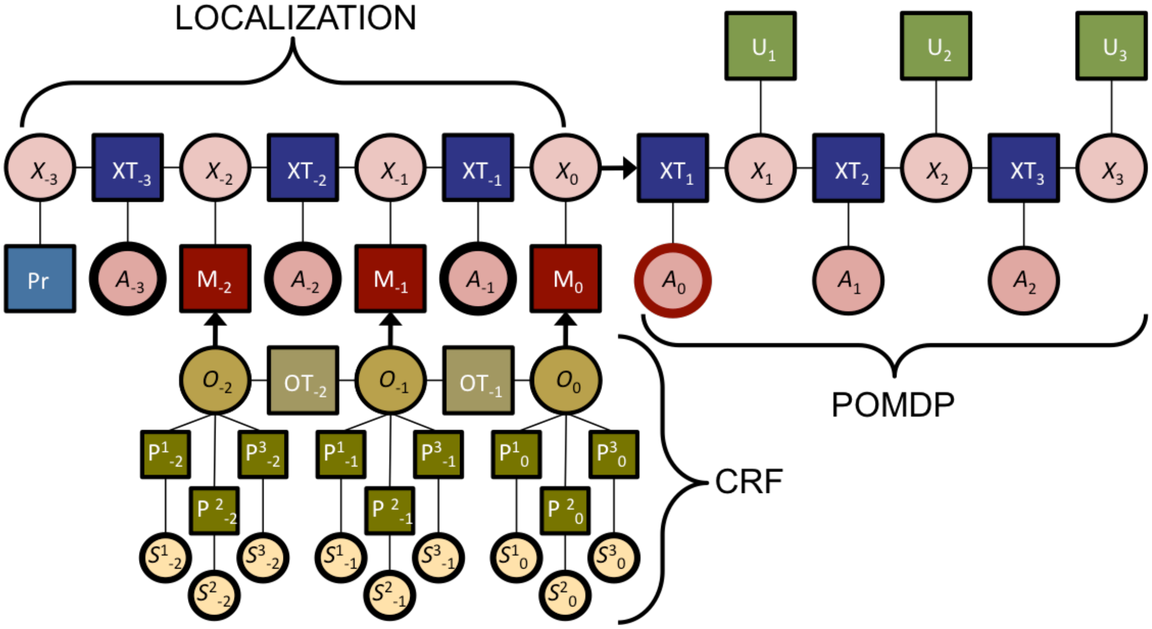


Figure 17: Combining a conditional random field, a probabilistic localization network, and a POMDP as factor graphs in a composite graph that spans from perception to decision making (Chen et al., 2011).

Section 4.4, but in summary at this conceptual level, such memory structures in Sigma have to date been shown to subsume – typically via specific cognitive idioms – forms of procedural (rule), declarative (semantic and episodic), constraint, perceptual (speech and vision), action and imagery memories, along with various combinations of them. Memory retrieval, basic reasoning, and perceptual processing all occur in Sigma by the processing implied by such structures.

4.2 Adaptation Phase

The adaptation phase is inherently about making changes. This includes decisions about the operators to be applied to make progress in the current situation, but it also includes changes to working memory more generally plus aspects of reflection, learning, affect (i.e., emotion), and attention. This material is sequenced as follows in the rest of this section: selection and changes to WM (Section 4.2.1), goals (Section 4.2.2), affect/emotion (Section 4.2.3), learning (Section 4.2.4), surprise (Section 4.2.5), and attention (Section 4.2.6). Although affect is listed as the topic of Section 4.2.3, a full comprehension of it and its implications in Sigma requires an understanding of much of the rest of this overall discussion of the adaptation phase. Also, although attention is listed as the topic of Section 4.2.6, much of the relevant material is by necessity delayed until the tri-level control structure is introduced in Section 4.3; and although reflection is mentioned in Section 4.2.1, its discussion is also not completed until Section 4.3.

4.2.1 Selection and Changes to WM

Operator choices in Sigma – for example, choosing an operator to print the current concept, as suggested by the rule from the classification scenario in Figure 16(b), or choosing to move the 3 tile down in Figure 11 – involve selecting a single value for the *Operator* variable of an architecturally distinguished WM function. When multiple agents are to be run simultaneously, this WM function also includes an *Agent* variable, with independent decisions then being made for each of the agents represented.

Rather than including a special language of preferences for such decisions, they are based on implicit numeric preferences, by choosing the *argmax* of the distribution over the *Operator* variable (Rosenbloom, 2011c). For the rule in the classification scenario, each possible concept is weighted according to the probability it is the correct concept, as computed by the classifier, with the most probable concept – `walker` in this case – being thus chosen as the operator. Operator selection of this sort, however, turns out to be just one special case of Sigma's general ability to select elements from distributions in updating working memory. Such selections can happen for arbitrary functions, and need not be based on argmax. In general Sigma can also select the *expected value* of a distribution; do *probability matching* to select one element with a likelihood that corresponds to its value in the distribution; do *Boltzmann selection*, corresponding to probability matching on a distribution that is first weighted and exponentially transformed; or *maintain the entire distribution*, as is, without an explicit choice. A key task of the elaboration phase is to yield the distributions that support these selections.

For operator selection, only a subset of these alternatives is valid, based on the implicit assumption that a single element is always to be selected if the distribution is sufficient to determine which element it should be. No operator is selected if none of them has a positive value, as a positive value acts much like an *acceptable* preference in Soar. No element is also selected if there are multiple elements with the same maximum positive value, but where this maximum value is less than 1. (A value of 1 is roughly equivalent to a *best* preference in Soar. So a random selection is made if there are multiple elements with a value of 1.) When no element is selected, an *impasse* is generated – implying that no direct progress can be made on the current decision – that triggers reflective problem solving during which indirect progress may be possible, as is discussed further in Section 4.3. An impasse can occur when no element has a positive value (a *none* impasse), when there are multiple non-1 maximal elements (a *tie* impasse), or when an operator remains selected for more than one cycle (a *no-change* impasse).

Beyond operator selection, changes to working memory can take a variety of shapes, the most traditional of which corresponds to state changes. In the problem space model, this concerns the (internal) application of operators, such as to change the locations of tiles in the Eight Puzzle. But not all changes to working memory need be driven by operator application, nor must they even involve selections. For example, in isolated word recognition, rather than using an explicit HMM that encodes a predefined number of time slices, as in Figure 14, only a single generic time slice may be directly encoded, as in Figure 18, with the results generated during each cognitive cycle yielding an updated distribution over the state in working memory for use during the next cycle. This enables forward processing in an HMM of unbounded length, over a deliberative sequence of cycles, with a selection – of the best word rather than the best operator – occurring only when the end of a word is detected.

When *prediction mode* is enabled, the state being built up in WM during the elaboration phase, such as S_i in Figure 18, is architecturally distinguished from the previous state, such as $S_{i\text{-}1}$ in the figure, to enable both to be accessed simultaneously. This lets $S_{i\text{-}1}$ and S_i coexist during the elaboration phase, and enables the architecture to automatically shift the contents of S_i back to $S_{i\text{-}1}$ during the adaptation phase. Prediction mode also supports the learning of transition functions, where access to successive pairs of states is essential. Although presently an option, prediction mode is likely to become the standard mode of handling state changes in the future. It is more broadly a step towards incorporating into Sigma the principle that prediction, and learning to predict, should be pervasive in cognitive systems (Bubic et al., 2010), and is, in fact, as central to the cognitive cycle as decision making.

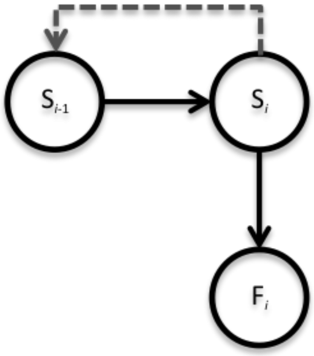


Figure 18: HMM with a single generic time slice to be reused each cycle.

4.2.2 GOALS

Goals in Sigma, and Soar before it, have had somewhat of an ambiguous status, despite both systems nominally being problem solving architectures. Impasses yield a form of architecturally generated – and thus architecturally penetrable – (sub)goal. Yet, although other types of more deliberate goals, such as for the desired configuration of tiles in the Eight Puzzle, can easily be expressed via knowledge above the architecture, and used in this form both to guide a search and to detect when it should be terminated, such goals are completely opaque to the architecture. The architecture is neither aware of such goals nor able to facilitate their achievement.

This conundrum was recently resolved in Sigma by partitioning working memory into normal *state functions* versus *goal functions*, with the latter being architecturally linked to the former (Rosenbloom, Gratch and Ustun, 2015). The architecture was also then extended to automatically compare corresponding state and goal functions, yielding measures of *similarity* (or *progress*) and *difference*. This involves – for each combination of values of the independent variables in the function, such as those like *X* and *Y* that are not normalized in the Eight Puzzle board function – comparing the distribution over the dependent (normalized) variables via the *Bhattarcharyya* *coefficient* (*BC*) to determine the similarity of that part of the goal (*G*) to the corresponding part of the state (*S*), and via the *Hellinger distance* (*HD*) to determine the difference between them:

$$Similarity(S,G) = BC(S,G) = \int \sqrt{s(x)g(x)}\,dx. \quad (1)$$

$$Difference(S,G) = HD(S,G) = \sqrt{1 - BC(S,G)}. \quad (2)$$

These measures yield new functions in which the original dependent variables (e.g., *Tile*) have been eliminated and in which a form of normalization – division by the integral over the goal function – is applied over what originally were the independent variables (e.g., *X* and *Y*).

There is a large space of measures that could have been considered for comparing distributions. The *dot product*, and its normalized variant the *cosine similarity*, can be used to provide measures of similarity for discrete distributions, even though they are technically applicable to vectors rather than distributions. The *Kullback–Leibler (KL) divergence* is a common measure for the difference between two distributions, with KL divergence and the Hellinger distance in fact being just two instances of a larger generic space of *f-divergence* measures of the distance between two distributions. The Bhattarcharyya coefficient also yields, in a different way from the Hellinger distance, the *Bhattarcharyya distance*. The Hellinger distance was chosen for Sigma, rather than the KL divergence or the Bhattarcharyya distance, primarily because it can handle 0 values in distributions, which are common in Sigma but problematic for both of the other measures. The Hellinger distance is also symmetric, and thus provides a metric in contrast to the KL divergence, and it obeys the triangle inequality, in contrast to the Bhattarcharyya distance. With the Hellinger distance, the computation of the Bhattarcharyya coefficient as a measure of similarity comes along for free.

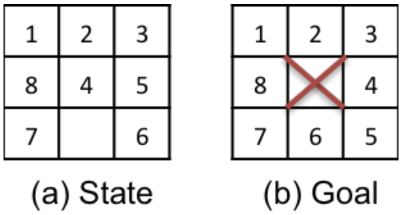


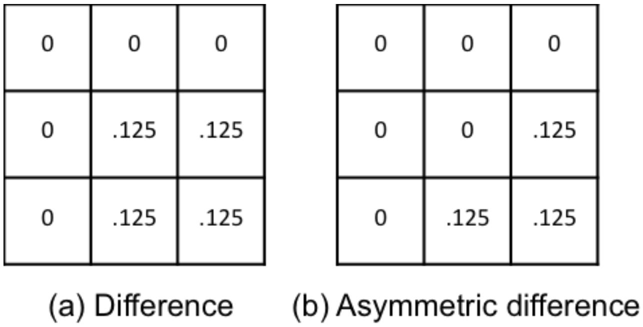
Figure 19: Example Eight Puzzle state and goal (there is no goal for the center cell).

As an example of the use of these measures, consider the Eight Puzzle state and goal in Figure 19, both of which are defined in Sigma as in Figure 12 in terms of Boolean distributions over the tiles given the (continuous) locations, where there is a 1 only for the correct tile at each location and a 0 for all others. There are eight nonzero regions of the goal here – as no goal has been stated for the blank – so that after normalization all of the entries are either 0 or 1/8 = .125, as shown for similarity/progress in Figure 20. The rationale for this form of normalization is that if we were then to aggregate – that is, *integrate* – over such a function, the total similarity would be 1, and the difference 0, when the goal is reached. As a result, the aggregated similarity/progress function is directly usable as an architecturally derived, albeit rather weak, evaluation function for the Eight Puzzle.

.125	.125	.125
.125	0	0
.125	0	0

Figure 20: Eight Puzzle similarity function.

When there are Boolean goal and state distributions, as in the Eight Puzzle, what is computed for similarity/progress corresponds to the fraction of the goal conjuncts achieved. If instead there were a Boolean goal and a more general state distribution, what is computed would more closely correspond to the probability that the goal has been achieved. A full distribution over the goal would correspond more to a utility or heuristic function than a traditional goal. For example, one aspect of the evaluation function for the two-person board game *Othello* (or *Reversi*) – a territory capture game like *Go* but simpler – is called *weighted squares*, where discs placed in different locations, such as the corners or edges of the board versus the interior, have different heuristic values (Rosenbloom, 1982). When there is instead a full distribution over both the goal and state, the result should correspond more to probabilities of achieving utilities. Not all of these particular variations have yet been explored in implemented tasks, but they all should be realizable.



0	0	0
0	.125	.125
0	.125	.125

(a) Difference

0	0	0
0	0	.125
0	.125	.125

(b) Asymmetric difference

Figure 21: Eight Puzzle difference function. The asymmetric version sets the value to 0 wherever the goal isn't defined.

The difference function for the Eight Puzzle given the state and goal in Figure 19 is shown in Figure 21(a). In the *asymmetric version*, as shown in Figure 21(b), which is what is actually made available to Sigma, differences are set to 0 for regions with unspecified goals. This function could conceivably be used to drive means-ends problem solving, as in GPS (Newell et al., 1959), by identifying those goal conjuncts that aren't currently met.

Both similarity/progress and difference are forms of *architectural self-monitoring*, and thus appear in the perceptual buffer as proprioceptive inputs. Figure 22 shows a refined version of Sigma's cognitive memory structure that makes explicit both the split between states and goals in WM and the split between external perception and proprioception in PB.

4.2.3 AFFECT/EMOTION

Perhaps somewhat surprisingly, the initial motivation for the architectural approach to representing goals and comparing them to states that was described above, was to support the needs of *emotion*, or *affect*. Cognitively, affect concerns both the processing that leads to emotional states and the ways these emotional states impact thought and behavior. Affect is a central concern in interactions between humans and virtual humans: without it visually realistic virtual humans stray far into the Uncanny Valley while also losing much of their intended effectiveness. Affect is also likely critical in any autonomous system that is to survive and thrive in sufficiently complex and open physical and social environments. As mentioned earlier, in humans it captures part of what can be termed the *wisdom of evolution*, providing non-voluntary pressures that have proven via natural selection to be adaptive over the long-term.

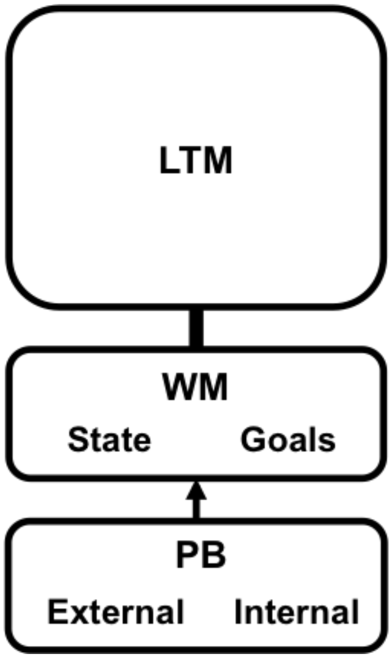


Figure 22: Refined structure of the cognitive memories.

The work on emotions in Sigma has so far focused on how both the induction of emotions and their impact on thought and behavior can be grounded in the architecture. This narrows the view for now to lower level aspects of emotion, deemphasizing those aspects dependent on knowledge and reasoning, which are left for future work.

The similarity/progress and difference functions comprise part of an architecturally grounded approach to *appraisal* (Moors et al., 2013), what is typically considered the first stage in the arc of emotional processing. During appraisal the current situation is assessed to yield values that feed into the development of emotional states and responses. Similarity/progress and difference map onto the *desirability* appraisal variable in affective models such as EMA (Marsella and Gratch, 2009). Sigma also embodies a variant of the *expectedness* appraisal variable, but its introduction will be deferred until Section 4.2.5, after learning is discussed in Section 4.2.4, because its computation depends on learning models that can generate expectations. Both forms of appraisal also impact attention, as will be discussed in Section 4.2.6.

4.2.4 LEARNING

Learning – that is, changes to long-term memory – occurs in Sigma via a process of *gradient descent* over its functions (Rosenbloom et al., 2013). This can lead to learning distributions over parameters as well as learning values of relations, but it does not acquire or modify the more complex structures that are also found in LTM. Sigma is currently unable to learn new structures, including anything corresponding to Soar's chunking mechanism (Laird et al., 1986; Rosenbloom, 2006). To understand the source of the gradients for learning, one must be familiar with the graphical architecture – as gradients are messages arriving at the factor nodes that store the LTM functions – and so this aspect will not be explained until Section 5.3. However, we can outline here how these gradients are used to update LTM functions.

For functions that use standard normalization – where the integral over the dependent (or *child*) variable(s) equals 1 – the learning algorithm is an adaptation of one earlier developed for Bayesian networks (Russell et al., 1995), but with smoothing – that is, small increases to parts of the updated function that ensure no element is below a specified threshold – and other adjustments made to it. The essence of the algorithm, minus the final smoothing (and possible post-normalization) operations, is captured by:

$$g_t'(\vec{v}) = {g_t(\vec{v})} \Big/ {\int f_{t-1}(\vec{v}) g_t(\vec{v}) d\vec{v}} \quad (3)$$

$$f_t(\vec{v}) = f_{t-1}(\vec{v}) + scale(\lambda(g_t'(\vec{v}) - average(g_t'(\vec{v})))) \quad (4)$$

Here, $\vec{v}$ is the vector of dependent/child variables; $g_t(\vec{v})$ is the incoming gradient; $g_t'(\vec{v})$ is this gradient divided by the integral over the product of the incoming gradient and the *prior* function $f_{t-1}(\vec{v})$; λ is the learning rate; and $f_t(\vec{v})$ is the result (or *posterior*) function. The average is subtracted from the adjusted gradient before it is used to update the function so that it integrates to 0, and thus so that the new function still integrates to 1 and follows the constraint surface. To avoid wild swings, the gradient is capped by scaling it down multiplicatively if its maximum is above a threshold. Key parameters to this algorithm include the *learning rate*, the *smoothing threshold*, the *maximum increment* and the form of *normalization*.

This algorithm has also been explored for learning functions in which no variables are normalized, such as in learning constraints, but a more appropriate algorithm is ultimately needed for these types of undirected functions. For functions that use vector normalization – that is, ones representing distributed vectors – gradient descent is simpler, amounting to multiplying the incoming gradient by the learning rate and adding the result to the original function:

$$f_t = f_{t-1} + \lambda g_t \quad (5)$$

This corresponds to how learning works in cognitive science models based on distributed vectors, such as BEAGLE (Jones and Mewhort, 2007).

True to functional elegance, this simple learning algorithm underlies distribution tracking, concept learning (both supervised and unsupervised) – as for the classifier in the classification scenario (Figure 7) – reinforcement learning, inverse reinforcement learning, learning of action models (in reinforcement learning), learning of transition and perception models (in HMMs, such as the one in Figure 14 for speech recognition), learning of maps (in SLAM), multiagent learning, and more (Rosenbloom, et al., 2013; Joshi et al., 2014; Pynadath et al., 2014; Rosenbloom, 2012a, 2014). Each of these higher-level types of learning effectively induces a cognitive idiom comprised of some knowledge plus gradient descent. Via a further application of functional elegance, when multiple such idioms end up being isomorphic to each other, they are merged into a single more general idiom. Two prime examples of this are learning action models in RL versus learning transition models in HMMs, and learning perception models in speech versus learning maps in SLAM (Rosenbloom, 2012a; Rosenbloom et al., 2013; Joshi et al., 2014). More on learning idioms can be found in Section 6.2, but to provide one concrete example here, Figure 23 shows how the learned map for the simple 1D world of Figure 10, which is expressed in terms of probability distributions for objects given locations – **P**(*Object* | *X*) – is refined over time.

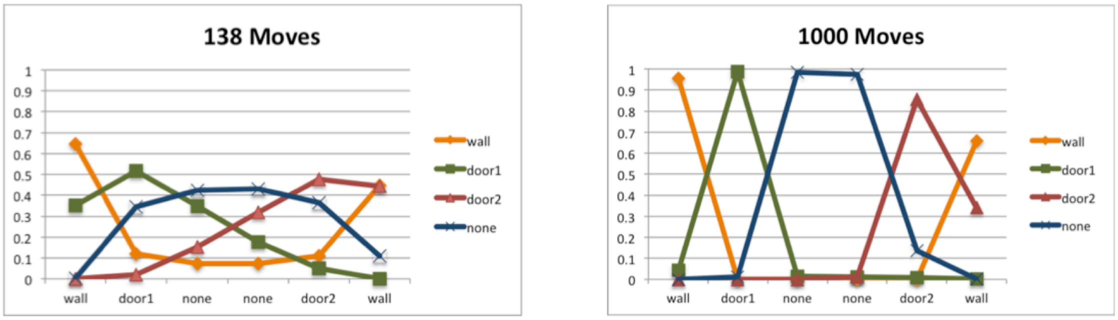


Figure 23: Results from map learning in Simultaneous Localization and Mapping (SLAM).

4.2.5 SURPRISE

During the process of updating a function according to a gradient, there is a point during the adaptation phase just before the prior is replaced by the posterior at which they co-exist, and which therefore provides a window of opportunity for leveraging them in computing a measure of *surprise* in a manner similar to the *Bayesian theory of surprise* (Itti and Baldi, 2006). In this model the prior, which reflects the best understanding of the function before the latest input, is compared via KL divergence to the posterior (which takes the input into account). The better the prior predicts the input, the smaller the change is required in moving to the posterior, enabling their difference to be used directly as a measure of surprise. In Sigma, the Hellinger distance, as earlier discussed in the context of desirability, is used instead of KL divergence; and the existing gradient-descent learning algorithm is used to update the model, as opposed to Bayesian belief updating. The result directly yields the *(un)expectedness* appraisal variable mentioned in Section 4.2.2.

4.2.6 ATTENTION

The last capability incorporated within the adaptation phase in Figure 6(b) is *attention* (Frintrop et al., 2010; Itti and Borji, 2013). This, along with affect, represents some of the most recent work in Sigma, and thus also some of the most preliminary and incomplete. Attention broadly concerns the effective allocation of limited resources. What isn't always recognized, though, is that it in general can be pertinent at any of the three levels of control – reactive, deliberative and reflective – although not necessarily in a uniform manner across levels. Because of this stratification, the detailed discussion of attention in Sigma is deferred until Section 4.3.1.

4.3 Nested Tri-Level Control Structure

In Sigma, a single cognitive cycle – comprised of input and output plus one elaboration phase and one adaptation phase – directly provides a *reactive capability* (Figure 8(a)). Activities such as classifying an object (Figure 7), recognizing a fragment of speech (Figure 14), solving the Ultimatum Game (Figure 15), and executing a body of rules (Figure 16) can all occur reactively, within a single cognitive cycle. The entire classification scenario – from perception through classification, selection, and action – occurs reactively in this manner, within a single cognitive cycle. In human modeling, a single cognitive cycle may be sufficient for a simple reaction time task, where a signal leads to an immediate choice of a response. Reactive processing is intended to be computationally limited – although it isn't at present in Sigma – and largely monotonic.

A *deliberative capability* uses this reactive capability, and thus individual cognitive cycles, as its inner loop in providing sequential, knowledge-driven – or algorithmic – behavior (Figure 8(b)). Cognitive cycles enable both the intelligent selection of operators – as in the classification scenario – and their application to yield new states, with deliberation then capable of stringing together an arbitrarily long sequence of such steps to provide potentially unlimited, and largely non-monotonic, computation within a single problem space. When sufficient knowledge is available to directly select operators in the Eight Puzzle without impassing, it can be solved in a deliberative fashion, with a sequence of operators being selected and applied that transforms the initial state of the board into a goal configuration (Figure 24). Although there may be no explicit goal in SLAM – the goal of learning a map is often implicit – a random walk driven by valuing all possible moves at 1 can occur at the deliberative level as well.

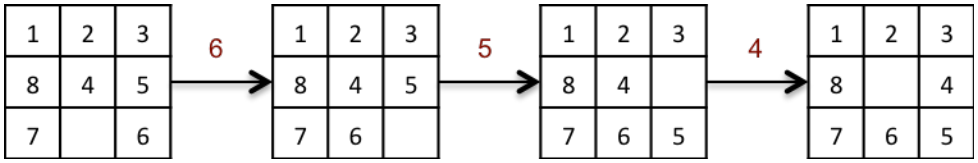


Figure 24: Deliberative solution of an Eight Puzzle problem over 7 cognitive cycles, 3 each for selecting and applying operators, and 1 to detect the goal and halt.

A *reflective capability* (Maes and Nardi, 1988) in turn uses this deliberative capability as its inner loop (Figure 8(c)). In particular, when impasses occur, (sub)goals are automatically generated, along with accompanying metalevels in which deliberative processing – either in the same or different problem spaces – can be used reflectively to yield the knowledge that would resolve the impasses. For our purposes here, reflection can be considered to comprise reasoning about reasoning. At the *base level*, reasoning is about the world. At the first metalevel, reasoning is about the base level; at the second metalevel, reasoning is about the first metalevel (and possibly the base level); and so on up the metalevel hierarchy. In Sigma, as in Soar

(Rosenbloom et al., 1988), reflection and new metalevels are initiated when an impasse occurs in processing at some level that makes progress at that level impossible.

Figure 25, for example, shows what happens if the control information is eliminated from the Eight Puzzle that enables the straight-line solution in Figure 24. Tie impasses result at the base level, which are approached as metalevel (sub)goals to be resolved by trying out the operators in simulation to see how good they are. In this case, the result is a form of steepest-ascent hill climbing across meta-levels, as (meta-)operators are selected within the tie impasse to evaluate the original operators. This yields no-change impasses within which the operators are tried and the evaluations of the resulting states – based either on explicit evaluation knowledge or on architecturally generated desirability appraisals – are assigned to the operators.

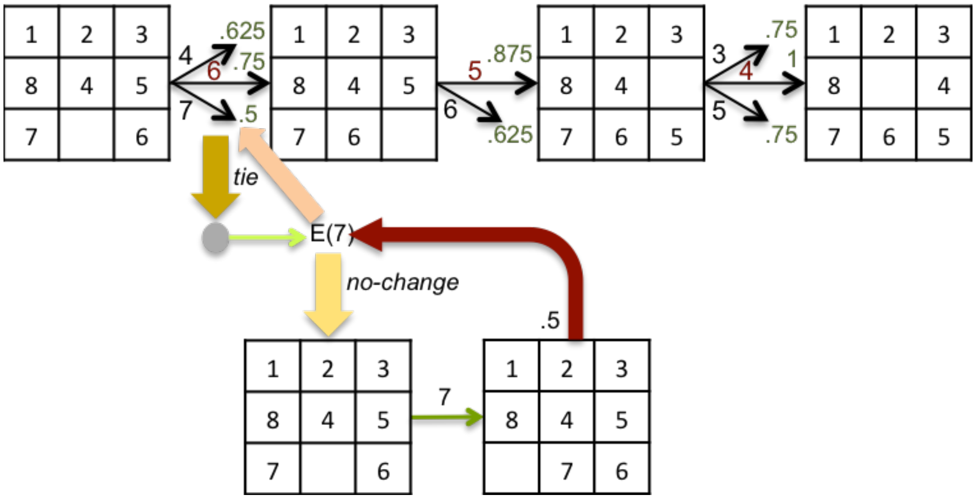


Figure 25: Reflective hill climbing within Sigma for the Eight Puzzle. Shown are the two metalevels for evaluating sliding the 7 tile at the first decision, plus the evaluations yielded by this kind of lookahead for each option at each decision.

To support the possibility of reflective processing, functions in Sigma may have an additional variable added for the *State*, as in **P**(*Tile* | *State*, *X*, *Y*) for the Eight Puzzle board. Rather than representing a state in a problem space, the *State* here represents the metalevel via a discrete number from 0-100. The base level is state 0, with each higher metalevel (or lower subgoal) assigned a number one more than its predecessor. The use of the term *state* for levels of the reflective hierarchy in Sigma matches the corresponding usage in Soar, but it also thus retains the potential confusion as to whether higher numbered states represent later situations in a problem space search. They do when search occurs across impasses, but not when it merely involves application of operators to states at one level. In the latter situation, the state number remains unchanged as progress is made through the space.

A more complex multiagent example can be seen in Figure 26, which provides a schematic for reflective decision-theoretic problem solving in the Ultimatum Game introduced in Section 4.1.2 (Pynadath et al., 2013). The impasse and metalevel/subgoal structure here is very similar to that of the Eight Puzzle, but evaluation is possible only at end states – based on the rewards in the game tree (Figure 27) – so lookahead proceeds to the end of the game at each top-level tie impasse via deep metalevels/subgoals. Between Figures 15 and 26 it should be clear that a decision-theoretic task such as solving the Ultimatum Game can be accomplished either in a reactive or a reflective manner (and possibly even in a deliberative manner, but that has not yet been investigated). Comparable results are generated via the two approaches, but the reactive approach is much faster, while the reflective approach is much more flexible. In Soar, chunking

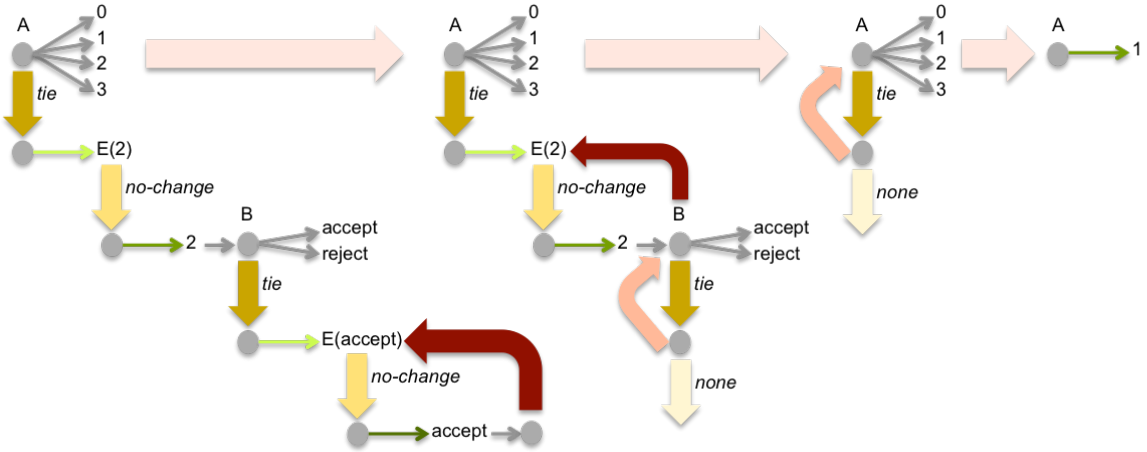


Figure 26: Reflective decision-theoretic problem solving within Sigma for the Ultimatum Game.

yields a skill acquisition mechanism that compiles (symbolic) reflective processing into (symbolic) reactive processing (Laird et al., 1986). In Sigma, it is hoped that a planned generalization of chunking will not only be able to compile symbolic processing, but also processing that is decision-theoretic (as with the Ultimatum Game here) or even perceptual (such as compiling the HMM in Figure 14 from some form of reflective processing).

Sigma's overall tri-level control structure maps well onto a number of existing bi-level and tri-level conceptions of cognitive processing, although the nesting of these levels is unique to Soar and Sigma. For example, the reactive capability maps not only onto the notion of reactivity, but also *automatic* (Schneider and Shiffrin, 1977), and *System 1* (Kahneman, 2011) processing. The deliberative capability maps onto *knowledge intensive*, *algorithmic*, and *controlled* (Schneider and Shiffrin, 1977) processing. Some combination of the deliberative and reflective capabilities maps onto *System 2* (Kahneman, 2011) processing. The overall tri-level structure also maps directly onto tri-level control structures that have been proposed in robotics (Bonasso et al., 1997) and emotion (Ortony et al., 2005). In traditional computer science terms, the reactive level provides a parallel control structure, the deliberative level a sequential or iterative control structure, and the reflective level a subroutine or recursive control structure (with the *State* variable providing the call stack).

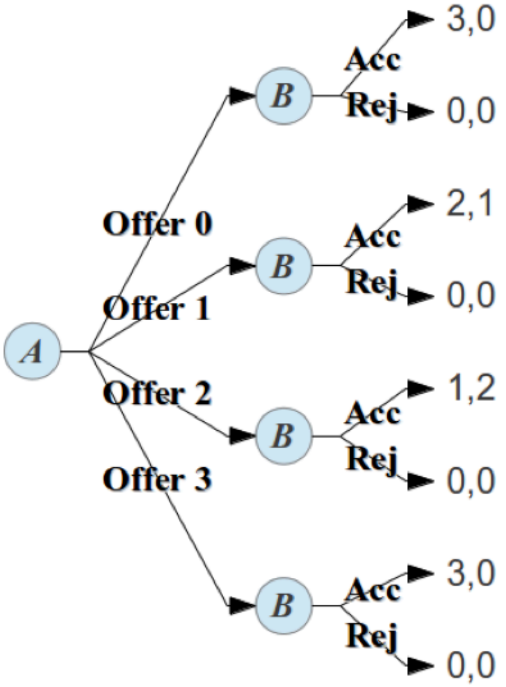


Figure 27: Reward tree for the Ultimatum Game, showing the rewards received for each combination of an offer and a response to it.

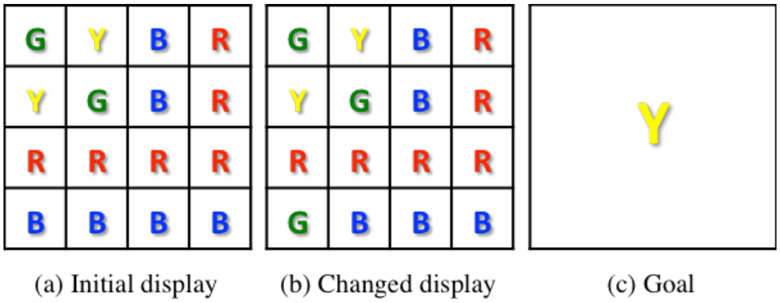
The combination of the reactive and deliberative capabilities can also be seen as a generalization of the control structure found in the *expectation maximization* (*EM*) *algorithm* for finding maximum likelihood parameters of statistical models (Dempster et al., 1977). A single cycle of EM generates expectations given the current parameter values and then modifies these parameter values given the expectations. These two steps map abstractly onto the elaboration and adaptation phases within Sigma's cognitive cycle, and thus to a single reactive step. The full algorithm involves a sequence of such steps, mapping onto deliberation. The claim here is not that Sigma subsumes the EM algorithm, as there are mismatches in the details, but that EM can be seen as a particular instantiation of the generic reactive+deliberative control structure, perhaps ultimately enabling EM to be placed more in context with a complete model of cognitive processing.

4.3.1 LEVELS OF ATTENTION

As mentioned in Section 4.2.6, attention, which concerns the allocation of limited resources, is relevant across all three of Sigma's control levels. Reflective attention is based on impasses; that is, each impasse focuses the metalevel processing that happens in response to it. In fact, the recent work in Sigma on appraisal, and its relationship to attention, has led to the conclusion that the detection of impasses should itself be considered as a form of appraisal, likely related to the notion of *controllability* in appraisal theories (Marsella and Gratch, 2009). Deliberative attention is based on the decision process, and the knowledge that feeds into it, as this determines which operators – and which values from other distributions – are selected to direct behavior. Thus, even before work was explicitly begun on attention in Sigma, it already embodied two architectural attention mechanisms (with the same also being true of Soar, with which it shares these mechanisms).

What has been missing in Sigma, and in most other cognitive architectures (except see, for example, Madl and Franklin, 2012), with respect to attention is an appropriate form of it at the reactive level that spans both covert perceptual attention – such as when attention is redirected in the visual field without physically moving the eyes – and automatic forms of cognitive attention. However, Sigma now exhibits such a capability via a mechanism that selectively abstracts functions to make them simpler and thus smaller and more efficiently processed. This is based on a combination of bottom-up and top-down inputs, a key concept in complete models of perceptual attention (Itti and Borji, 2013). What is more, both of these types of inputs are derived from appraisal variables already described, implying that reactive attention is driven by a subset of the architectural assessments that drive emotional behavior. Bottom-up attention is based on surprise, a standard practice at this point in state-of-the-art attention models (Itti and Baldi, 2006), but here it is derived from the (un)expectedness appraisal variable and is applicable to any function that is learned, not just to perceptual functions. Top-down attention is typically assumed to involve goal or task relevance, but how this works in general is less standardized. In Sigma, the desirability appraisal variable relates aspects of the state to the corresponding aspects of the goal, thus providing an appropriate top-down input. These bottom-up and top-down factors are then combined via an approximation to *probabilistic or* that assumes independence between the two factors:

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B) \approx P(A) + P(B) - P(A)P(B). \qquad (6)$$



G	Y	B	R
Y	G	B	R
R	R	R	R
B	B	B	B

(a) Initial display

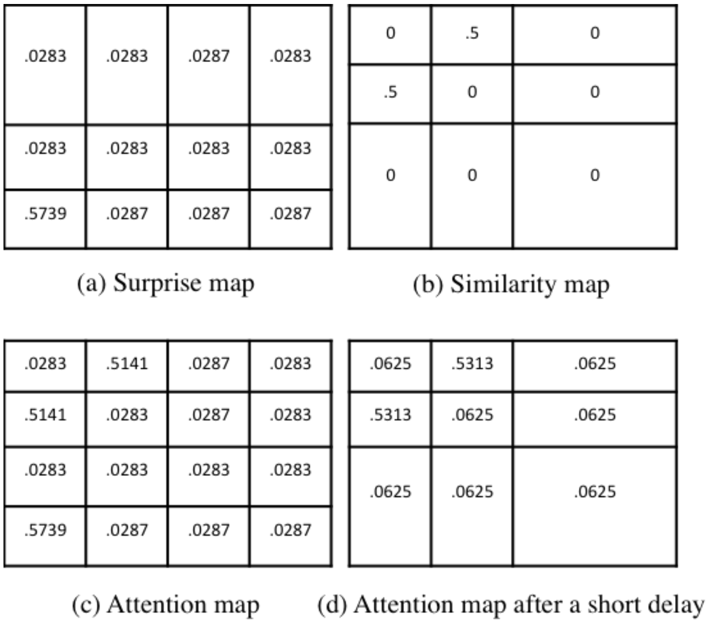
G	Y	B	R
Y	G	B	R
R	R	R	R
G	B	B	B

(b) Changed display

(c) Goal

Figure 28: Displays and goal for a visual search task.

Consider for example a visual search task, in which the goal is to find the yellow objects in the visual display shown in Figure 28(b) after first having stared for a while at the display in Figure 28(a). There is a bottom-up aspect to attention here, due to the surprise resulting from the change in the bottom-left corner from blue to green. There is also a top-down aspect due to the task of finding yellow objects (Figure 28(c)). The resulting bottom-up surprise/unexpectedness map can be seen in Figure 29(a), the top-down progress/similarity map in Figure 29(b), and the combined attention map in Figure 29(c). The influence of both factors can clearly be seen here, but in this particular case the bottom-up input outshines the top-down. After a bit more time, when the surprise dies down, the top-down influence regains ascendancy, as shown in Figure 29(d).



.0283	.0283	.0287	.0283
.0283	.0283	.0283	.0283
.5739	.0287	.0287	.0287

(a) Surprise map

0	.5	0
.5	0	0
0	0	0

(b) Similarity map

.0283	.5141	.0287	.0283
.5141	.0283	.0287	.0283
.0283	.0283	.0283	.0283
.5739	.0287	.0287	.0287

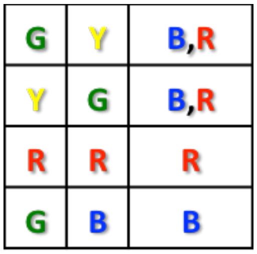
(c) Attention map

.0625	.5313	.0625
.5313	.0625	.0625
.0625	.0625	.0625

(d) Attention map after a short delay

Figure 29: Computed maps in visual search.

To use such a map in abstracting a function, it is scaled, exponentiated – to increase the contrast between regions with high and low attentional values – and then multiplied times the original function. When the differences between the distributions in adjacent regions of this result are small enough, the original regions are combined into a larger region whose function is the average of the regions out of which it is composed. For the display in Figure 28(b) and the attention map in Figure 29(c), this yields Figure 30, which has 12 rather than 16 regions. A similar experiment with a 200×200 display has shown a reduction from 160,000 regions down to 12, accompanied by savings in processing time.



G	Y	B,R
Y	G	B,R
R	R	R
G	B	B

Fig. 30: Abstracted function with two mixed blue-red cells.

4.4 Cognitive Language

Sigma's cognitive language is an instance of the general class of *statistical relational languages* (or *probabilistic programming languages*) (Milch et al., 2007; McCallum et al., 2008; Goodman et al., 2008; Domingos and Lowd, 2009). However, many of the choices in the language are constrained directly by the cognitive architecture – which was itself inspired by the graphical architecture hypothesis – rather than being free parameters. Consider for example a contrast with Alchemy (Domingos and Lowd, 2009), which is based on a clean conception of a probabilistic extension to first-order logic (in particular, Markov logic). Some pre-Sigma explorations of graphical cognitive architectures were pursued in Alchemy, and some aspects of Sigma's current language design still owe a debt to it; however, the inherent intractability of such logics suggests that it is not a good candidate for the inner reactive loop of a cognitive architecture. In fact, it can be argued that Alchemy makes a fundamental mistake in not being decomposed into reactive and deliberative layers at least, so that the more combinatoric aspects of theorem proving can be controlled by further knowledge (Rosenbloom, 2011a). Church (Goodman et al., 2008) provides a contrasting example, in which probabilities are added to an efficient procedural language (Scheme). The core problem here is that in focusing on a standard procedural language Church loses touch with much of what has been learned about cognitive architectures over the past few decades.

At its core, Sigma's cognitive language is based on *types* that specify the domains of variables, *functions* defined over combinations of variables (as in Section 4.1.1), *predicates* that specify relations among sets of typed arguments, and *conditionals* that structure long-term memory. Together these constructs provide a language that effectively and deeply merges ideas from both rule systems and probabilistic networks to yield a functionally elegant hybrid mixed language for encoding the knowledge and skills required to approach grand unification and generic cognition. Sufficient efficiency is approached by compiling the language down to the graphical architecture (Section 5.4), which is based on the state-of-the-art formalism of factor graphs but which itself must be implemented efficiently in a lower-level language such as Lisp (a topic omitted in this paper, but which is discussed in Rosenbloom, 2012c; Rosenbloom et al., 2015). The remainder of this section explains the four core constructs that comprise the cognitive language, with conditionals being by far the most complex.

4.4.1 TYPES

Types specify the domains of variables, including their scope, whether they are numeric or symbolic, and if they are numeric whether they are discrete or continuous. For instance, in the Eight Puzzle example, two types are used. The `tile[0:9)` type is discrete (`:`) over the integers 0-8 to represent the blank, as 0, and the eight numeric tiles – numeric ranges in types are always half open (closed at the bottom and open at the top).[3] The `dim[0-3)` type is similarly continuous (`-`) from 0 up to just less than 3. Two numeric types are also used in the classification scenario: `d4[0:4)` for the integers 0-3, as needed for the number of legs, and `weight[0-500)` for any weight below 500 (pounds). Four symbolic types are also used in this example: `boolean[false true]` for whether the object is either alive or mobile, `id[o1 o2 o3 o4 o5 o6]` for identifying up to six distinct objects at once, `color[silver brown white]` for object colors, and `category[walker table dog human]` for the concepts that can be assigned to objects. Although these symbolic types are relatively small, very large symbolic ones can also occur; for example, in language processing there may be a type containing all of the (thousands to hundreds of thousands of) words in a language.

[3] Given that Sigma is written in Lisp, Lisp-like representations are used in this article as appropriate, but more concise representations are also used, as here, when they improve simplicity and clarity. In these latter cases, the content is the same as is in Sigma's Lisp structures, but the syntax differs.

4.4.2 FUNCTIONS

Functions are just what were introduced in Section 4.1.1, but in the cognitive language they are defined over sets of typed variables. For example, the hybrid Eight Puzzle function in Figure 12 is defined over the `x`, `y`, and `tile` arguments, with the first two being of type `dim` (continuous) and the last one being of type `tile` (discrete). An occupancy grid for the state in Figure 11 may be specified as a function:

```
([0-1) [0-1) 1) ([0-1) [1-2) 4) ([0-1) [2-3) 7)
([1-2) [0-1) 2) ([1-2) [1-2) 5) ([1-2) [2-3) 8)
([2-3) [0-1) 3) ([2-3) [1-2) 0) ([2-3) [2-3) 6).
```

Here each triple defines a 3D region – that is, a 3D *orthotope* or *hyperrectangle* – with the assumption that a constant function of `1` is assigned to each region explicitly listed, with all other regions being assigned by default a constant function of `0`. If two regions overlap, the one specified last (re)defines the overlapping portion. If a constant function other than `0` or `1` is desired, the value can be placed in front, as in `.8:(o1 walker) .2:(o1 dog)` for a distribution over categories given object `o1`.

If a linear function over one of the variables is desired, the constant portion of the function remains in front of the region but the coefficient is added in front of the variable's scope, as in `.3:(o1 .001:*)` for the function `.3 + .001×weight` over the region spanned by object `o1` and all weights (specified via `*`) in the classification scenario. In general, Sigma's cognitive language is capable of specifying *piecewise linear functions* that can be decomposed into a set of *n* dimensional regions, each with its own linear function. Constant region functions are just special cases of this, with globally constant functions themselves being further special cases where there is a single region. When convenient, any function may still be displayed in a more tabular format, as was the practice in the previous subsections.

4.4.3 PREDICATES

Predicates specify relations among sets of typed arguments. Depending on the included types, plus various optional annotations, predicates can be specified as symbolic or probabilistic or mixed (symbolic + probabilistic), and discrete or continuous or hybrid (discrete + continuous). For example, in the Eight Puzzle, the hybrid `Board(x:dim y:dim tile:tile)` predicate defines a relation over the `x`, `y`, and `tile` arguments, where the first two arguments are of type `dim` (continuous) and the last is of type `tile` (discrete). In the classification scenario, if we combine the `id` and `category` types we can create a predicate for object classes: `Concept(object:id value:category)`. We could then extend this towards the full naïve Bayes classifier in Figure 7 by adding predicates for the five object attributes:

```
Alive(object:id value:boolean)
Legs(object:id value:d04)
Color(object:id value:color)
Mobile(object:id value:boolean)
Weight(object:id value:weight).
```

When combined with the appropriate functions for the relevant probability distributions plus the appropriate conditionals, as will be discussed shortly, a full classifier results that when given evidence of the values of cued attributes can retrieve (or predict) an object's category along with the values of its unseen attributes. The `value` arguments here essentially yield the variables of a Bayesian network, with the `object` argument inducing an instance of the network for each object to be classified.

Functions and predicates are closely related, but not in a one-to-one manner. First, each predicate may have multiple functions associated with it, as part of the three different forms of cognitive memory (Figure 9). In particular, functions can provide a learned model for a predicate in LTM, or the current state or goal of the predicate in WM, or what is being perceived about the predicate in PB. For example, the naïve Bayes classifier in the classification scenario (Figure 7) can be built from the predicates above plus additional predicates that relate concepts to features, with the appropriate conditional probabilities built into LTM functions for the predicates, as in

```
Concept-Mobile(concept:category mobile:Boolean
   &LTM   1:(walker true)
         .95:(table false) .05:(table true)
         .05:(dog false) .95:(dog true)
         .05:(human false) .95:(human true)),
```

where `&LTM` signals the beginning of the LTM function specification. Second, functions need not be tied to specific predicates. In particular, they may instead be associated with conditionals. Also, multiple predicates (or conditionals) may use the same function through *tying*, where they all point to the same function.

One key distinction among varieties of predicates is whether they are *unique* or *universal*, which is itself grounded in whether the predicate's arguments are unique or universal. Universal arguments are like variables in rule systems, where any or all of the elements in the variable's domain may be valid. Unique arguments are like random variables in probabilistic systems, where a distribution is provided over all of the elements of the variable's domain but only a single value is actually correct. In the naïve Bayes classifier, the `object` arguments are all universal, enabling any number of the objects to coexist, while the `value` arguments are unique. If unique arguments are normalized they may represent probability distributions, but they need not be normalized, nor if they are normalized must they represent probabilities. However, here they are normalized and do represent probabilities.

If all of the arguments in a predicate are unique then the predicate represents a full joint distribution, with the integral/sum over the entire distribution equaling 1 if it is normalized. If a subset of the arguments in a predicate is unique then the predicate represents a conditional distribution over this subset given the universal arguments. In the naïve Bayes classifier, for example, the `Concept-Mobile` predicate represents the conditional probability of mobility given the concept. Similarly, in the Eight Puzzle `Board` predicate the `tile` argument is unique, while the `x` and `y` arguments are universal, denoting that there is a single correct tile per location. No matter how many unique variables there are, there is a single correct value over the combination of them given any particular combination of values for whatever universal arguments there are. Any predicate that has at least one unique argument – which includes both the classifier predicates and the `board` predicate – is itself considered unique. All unique predicates can be mapped conceptually onto fragments of Bayesian networks, where the cross product of the unique arguments becomes a single *child* node and the universal arguments become its *parents*. In more mathematical terms, the cross product of the unique arguments is a function of the universal arguments.

An important sub-distinction for unique predicates concerns whether their WM functions maintain full distributions or embody selections of individual elements. Annotating a unique argument with `%` implies that the full distribution should be maintained; for example, the predicate definition `Concept(object:object value:category%)` in the classifier indicates that the full distribution over the category for each object should be represented in working memory. All of the feature predicates in the classifier also maintain the full distribution.

Selections, in contrast, imply choices must be made, as discussed in Section 4.2.1. As there are multiple ways to make such choices, there are multiple ways to annotate unique arguments. For example, use of `!` – as in `Board(x:dim y:dim tile:tile!)` for the Eight Puzzle – implies that a best alternative (i.e., the most likely tile) is to be selected; use of `$` denotes that the expected value of the unique argument is to be selected; and use of `=` denotes that probability matching should occur, where a single value is chosen randomly according to the distribution over the values.

Decisions in Sigma, in the classical sense of choosing one among the best operators to execute next, are an immediate consequence of introducing an architecturally distinguished selection predicate. In the rule for the classification scenario shown in Figure 16(b), the use of "Operator(*c*)" in the action was shorthand for `Selected(state:state operator:category!)`, where the `operator` argument is unique and defaults to random selection of one of the best alternatives (although this can be overridden, for example, to yield Boltzmann selection). Rather than representing states in a problem space, the `state` here represents metalevels (Section 4.3), in this case via a discrete numeric type: `state[0:100)`. Although the reasoning with the naïve Bayes classifier occurs reactively, so no impasses occur, the `state` argument is always there in case an impasse does occur. When there is no impasse, all activity occurs in state `0` (i.e., the base state).

Impasses are themselves also represented via a unique predicate – `Impasse(state:state operator:operator type:impasse!)` – that is based on a unique symbolic type: `impasse[none tie no-change]`. In multiagent situations, an additional `agent` argument supports agent-by-agent operator selection and impassing: `Selected(agent:agent state:state operator:operator!)` and `Impasse(agent:agent state:state operator:operator type:impasse!)`.

If there are no unique arguments in a predicate – that is, they are all universal – the predicate itself is considered to be universal. A universal predicate does not maintain a distribution; instead, each possible combination of values of its variables is considered independently of all of the others. Universal predicates map onto the traditional forms of relations found in rule systems, where each (potential) element of working memory is considered independently. The universal predicate `Above(first:object second:object)`, which for example would be used in the transitive rule in Figure 16(a), can represent a multi-valued relationship among objects. Universal predicates can also be mapped onto the Boolean-valued predicates traditionally found in logic.

A second key distinction among varieties of predicates is whether they are *distributed vectors*. In many ways a vector predicate behaves much like an unnormalized unique predicate, except that the former are normalized via standard vector normalization. In addition, as we have already seen, the way gradient descent works for them has been tuned; and, as we shall see shortly, the way actions are combined for them has been tuned as well.

A final key distinction among varieties of predicates concerns whether they are *open world* or *closed world*, and thus whether unspecified regions in their functions are assumed to be unknown (as in probabilistic networks and many logics) or false (as in rules and other logics). This distinction also ends up determining whether WM functions are recalculated for every decision or latched across decisions, thus mapping roughly onto the related notions of: *i-support* (elaboration) versus *o-support* (operator application) in Soar; justifications versus assumptions in truth maintenance systems; and theorems versus axioms in logic. Latching also maps onto the traditional notion of working memory in rule systems, where values remain until explicitly changed or deleted. Both Soar and Sigma go beyond this traditional rule notion to allow some elements in WM to be automatically retracted when no longer valid rather than requiring them to be explicitly modified or deleted. The Selected and Board predicates in the Eight Puzzle are both closed world, latching the operator and the state until they are explicitly changed. The predicates in the naïve Bayes classifier are all open world, with unspecified values assumed unknown and no latching in WM.

4.4.4 CONDITIONALS

Conditionals structure long-term memory in terms of interactions among *predicate patterns* – that is, *conditions*, *actions* and *condacts* (which yield bidirectional constraint by fusing aspects of both *cond*itions and *act*ion*s*, and which will be explained further shortly) – plus optional *functions*. They are named for how they provide a deep blending of the types of conditionality found in both rule systems (based on conditions and actions, as in Figure 31) and probabilistic networks (based on prior and conditional probability distributions, and condacts, as in Figure 32).

A predicate pattern looks much like a predicate definition, but it only includes the predicate's name and arguments, and argument types are replaced by constants (Roman font) or *variables* (italic font). The patterns in Figures 31(a) and 32 are all based on variables. In contrast, the pattern Concept(object:o1 value:*c*) in Figure 31(b) characterizes the whole distribution over the concepts for object o1; and Board(x:*x* y:*y* tile:1) characterizes the entire *xy* span of the Eight Puzzle board that is occupied by tile 1. Although it is possible, and often makes semantic sense, to use the same symbol for an argument and its associated type and/or variable – with only the context of use distinguishing them – different symbols can also be used when desirable. A pattern such as Concept(object:*o* value:walker) can, for example, be specified with the variable *o* for the object and the constant walker for the value.

```
CONDITIONAL Transitive-Above
   Conditions: Above(first:a second:b)
               Above(first:b second:c)
   Actions: Above(first:a second:c)
```

(a) Transitive conditional for *above*.

```
CONDITIONAL Select-Concept-as-Operator
   Conditions: Concept(object:o1 value:c)
   Actions: Selected(state:0 operator:c)
```

(b) Conditional for selecting the concept for object o1 as the operator in the classification scenario.

Figure 31: Conditionals for rules in Figure 16.

```
CONDITIONAL Concept-Mobile-CF
   Condacts: Concept(object:o value:c)
             Mobile(object:o value:m)
   Function(c,m):
```

	walker	table	dog	human
false	0	.95	.05	.05
true	1	.05	.95	.95

Figure 32: Conditional for conditional probability of an object being mobile given its concept. This provides one leg of the naïve Bayes classifier in Figure 7 for the multi-object case covered by the classification scenario.

A conditional that includes a function, as in Figure 32, must specify which pattern variables the function is defined over along with a function over these variables. In a Bayesian network, the direction of such a function's conditionality is determined by the directionality of the graph, with a child variable always conditioned on its parent variables. In predicate functions (PFs) the direction is determined by which arguments are unique versus universal. In conditional functions (CFs), the direction, which is necessary for gradient descent to correctly learn these functions, can be indicated by underscoring the child variables in the specification, as with *m* in the figure. If the functions in multiple conditionals are to be *tied*, then the actual function is provided for only one of them, while the others specify the name of this conditional instead.

Figure 33 shows an alternative way of encoding the same functionality found in Figure 32 but via an additional predicate that was introduced in Section 4.4.3 plus its associated function instead of a conditional function. Only conditional functions were supported in early versions of Sigma, but when it was realized that the specification of a conditional function essentially required the creation of a pseudo-predicate within the conditional – in terms of which pattern variables are involved and which of these are the children – the latter was added to make this simpler and more uniform. At this point, it is recommended that conditional functions be limited to constant functions – that is, ones without variables – with predicate functions taking over the heavy lifting when there are variables.

```
CONDITIONAL Concept-Mobile-PF
   Condacts: Concept(object:o value:c)
             Mobile(object:o value:m)
             Concept-Mobile(class:c mobile:m)
```

Figure 33: Alternative conditional for the computation in Figure 32, but based on a predicate function.

The examples in Figures 31-33 should not be taken to imply that rules and Bayesian networks are the only structural forms of knowledge available, nor that conditional probabilities are the only functions representable. The blending of concepts is at a deep enough level and at a small enough granularity that a substantially larger space of possibilities emerges. For example, Figure 34 shows a conditional for a link in a constraint satisfaction network that eliminates the possibility that vertices `v1` and `v2` are both `red`. Such a function is not inherently probabilistic. Functions also need not even involve variables, as already intimated. The conditional in Figure 35, for example, shows a rule that uses a function with a constant value of `1` to encode the equivalent of a *best* preference in Soar.

```
CONDITIONAL Constraint-C1-C2
   Condacts: Assign(vertex:v1 color:c1)
             Assign(vertex:v2 color:c2)
   Function(c1,c2):
```

	red	blue
red	0	1
blue	1	1

Figure 34: Conditional constraining two vertices in a graph to not both be `red`.

As shall be discussed further in Section 6.1, using conditions and actions together in conditionals, as in Figure 31, provides a procedural-memory idiom, whereas using condacts and functions together, as in Figures 32-34, provides a declarative-memory idiom. Thus, the use in Figure 35 of conditions, actions and a function yields a memory fragment that is in some sense a hybrid between these two. Other kinds of hybrids are also possible; for example, Figure 36 shows a conditional comprised of one condition, one condact and a function that provides an initial uniform distribution over a policy that is ultimately to be modified via reinforcement learning

```
CONDITIONAL Left-Best
   Conditions: Operator(id:left state:s x:x y:y)
               Board(state:s x:x y:y tile:t)
               Goal(state:s x:x-1 y:y tile:t)
   Actions: Selected(state:s operator:left)
   Function(): 1
```

Figure 35: Conditional in the Eight Puzzle that makes the `left` operator *best* if it moves a tile into its goal location.

```
CONDITIONAL Q
   Conditions: Location(state:s x:x)
   Condacts: Q(x:x operator:o value:q)
   Function(x,o,q): .1
```

Figure 36: Conditional for an initial uniform distribution over Q values for operators by location.

(RL) to yield improved operator selection.

Predicate patterns are closely tied to their predicates' WM functions. During conditional processing, a condition constrains to 0 all portions of the corresponding WM function that do not match the pattern. An action proposes changing the corresponding WM function to more closely match the pattern. In Figure 31(a), for example, the two conditions and the action all concern the `Above` WM function. The two conditions map the existing function onto variables *a* and *b*, and *b* and *c*, respectively, with the shared variable *b* constraining what results are legitimate for the combination of variables *a*, *b* and *c*. The action then proposes changes to the function based on variables *a* and *c*. Given `Above(first:o1 second:o2)`, `Above(first:o2 second:o3)`, `Above(first:o4 second:o5)` and `Above(first:o5 second:o6)` in the WM function, the conditional suggests that `Above(first:o1 second:o3)` and `Above(first:o4 second:o6)` be added to the `Above` WM function.

One important subtlety here is what happens to variable *b* in moving from the conditions to the action. In a standard rule system, an *a-c* pair would result whenever there is some value of *b* compatible with the pair. Thus there is an implicit existential quantifier for *b*. In Sigma this occurs by *summarizing out* variable *b* via *max* from the function over *a*, *b* and *c* yielded by the conditions; that is, by assigning to each *a-c* pair a value corresponding to the maximal value for that pair across all elements of variable *b*. For a Boolean function, such as here, this yields a 1 (i.e., *true*) for every *a-c* pair that has at least one *a-b-c* triple with a value of 1.

This approach was inspired by the summary product algorithm used in the graphical architecture, where variables are summarized out in messages either by sum/integration or max. In the cognitive architecture, all universal variables are summarized out via max. Unique variables are typically summarized out by sum/integration, to yield the *marginal distribution* over the individual variables; that is, the distribution over each variable that is yielded when all of the other variables are summarized out. However, unique variables can also be summarized out via max, to yield the best joint choice over all of the relevant variables (what is known in probabilistic reasoning as *maximum a posteriori* (*MAP*) estimation).

Both conditions and actions can be *negated* by inverting their functional values. In particular, inverting a function *f* yields a function $\overline{f} = \max(1-f,0)$. *True* (`1`) becomes *false* (`0`) and vice versa. Intermediate values are similarly inverted, while functional values greater than 1 are treated as if they are 1 during the inversion. Figure 37, for example, shows a degenerate (conditionless) rule that removes the first object from an image via a negated action that inverts the object's entire plane (with `x` and `y` specified implicitly as unnamed variables over their whole domains).

```
CONDITIONAL Delete-1
   Actions: -Image(o:1)
```

Figure 37: Deletion of object 1 from an image.

A condact exhibits effects that are akin to both a condition and an action. Like a condition it yields a constrained function for combination with the rest of the conditional. Like an action it proposes changes to WM based on the function determined by the rest of the conditional. Condacts effectively combine local constraint from the predicate's own segment of working memory with global constraint from both long-term memory and the rest of working memory, in support of, for example, partial matching in declarative memory, constraint satisfaction, signal processing, and general probabilistic reasoning. Figure 32, for example, specified a fragment of general probabilistic reasoning via two condacts and a function. The binding of variables *o* and *c* by the `Concept` condact combines with the function to constrain the `Mobile` WM, while simultaneously the binding of variables *o* and *m* by the `Mobile` condact combines with the function to constrain the `Concept` WM. With only conditions and actions, and no condacts, such influence could only proceed in one direction, either from the category to the feature or vice versa, but not in both directions. Likewise, in SLAM, the map could impact localization or it could be learned from perception of objects in particular locations, but both localization and mapping could not proceed simultaneously. Condacts have so far always been based on open-world predicates and typically include no constants, being instead constrained by conditional or predicate functions.

Modifications to working memory can be proposed by both actions and condacts, but with distinct differences in how they operate. Actions, when based as they usually are on closed-world predicates, change working memory in a persistent manner – latching their results – while condacts, which are usually based on open-world predicates, typically have no persistence. Actions for open-world predicates behave along this dimension like condacts, with no persistence. A distinction also exists between actions and condacts in what it means to combine several of them. In a sequential rule system there is no fundamental need to combine actions, as only one action is executed at a time. In a parallel rule system, such as Soar, multiple actions are combined disjunctively. Initially disjunction was mapped onto *maximum* in Sigma, but max is now only used in action combination for non-vector universal predicates. For unique predicates with normalized functions, actions are combined via the same approximation of *probabilistic-or* used for combining bottom-up and top-down inputs in attention (Equation 6).[4] Actions are combined via *summation* for both non-normalized unique predicates and vector predicates.

Disjunctive combination, as just described, is actually only used for positive actions. Negated actions for vector products are combined by *subtracting* them from the result of the positive actions. For all other negated actions, the inverted function is combined with the positive actions via *product*, so that regions are deleted if they are positively mentioned in any such negated action. Likewise, multiple condacts combine via product, as might be expected of constraints that are being conjoined. If non-vector positive actions exist along with either condacts or negated actions for the same predicate, the positive actions are first combined disjunctively, with the result then multiplied in with the condacts and negated actions.

walker	table	dog	human
1	0	0	0

Figure 38: Piecewise constant function for the constant `walker` in a `category` variable.

The earlier statement that arguments in predicate patterns are specified by constants or variables actually oversimplifies the situation considerably, as the language supports significant generalizations of the traditional notions of both constants and variables. Starting with the former, a constant in a condition denotes that the pattern should only match to those parts of working memory with that value. This functionality can be reconceptualized as a *piecewise constant filter* that has a value of 1 for the constant mentioned and 0 values for all of the others (Figure 38). Match of the pattern to the appropriate working memory then simply involves a *pointwise product* – like an inner or dot product but without the final summation over all of the elements – of the WM function times the piecewise constant function that represents the pattern filter. When an argument in a pattern is specified as a variable, it implies a value of 1 across the argument's whole domain.

Based on this perspective, which was inspired by the graphical architecture, it was realized that the full expressibility of *piecewise linear filters* was already supported by the architecture, and so the language was extended to allow this as well. For example, the condition `Q(x:x operator:o value:[.1*q])` in Figure 39 uses a linear filter with a slope of .1 and an

[4] This is problematic if values can be greater than `1`, so incoming messages are first scaled uniformly to a maximum value of `1` whenever their maximum value exceeds this.

intercept of 0 to weight the different possible Q values – that is, utilities of possible operators (*o*) given particular attributes of the state (*x*) – by their probabilities. In this example, the Q values range in [0,10), so using a coefficient of .1 scales this down to [0,1). Using this condition enables the action for the Selected predicate to rate operators according to their expected Q values for the current location, so that a decision can be based on a learned policy. In particular, the variable *q* here is summarized out before the action is executed, and because value is a unique argument in the Q predicate, this involves integrating over the (scaled down) Q values, as weighted by their probabilities, to yield a functional value for each operator corresponding to its expected Q value.

```
CONDITIONAL Select-Operator
   Conditions: Location(state:s x:x)
               Q(x:x operator:o value:[.1*q])
   Actions: Selected(state:s operator:o)
```

Figure 39: Conditional that transforms a policy – a distribution over utilities in [0,10) for operators given locations – into operator weights – in [0,1) – for selection.

The generalization for variables supports *affine transforms* over them; that is, combinations of linear transforms and translations that can together yield object translations, rotations, scaling and reflections. Such transforms are central to work in mental imagery (Section 6.1.4), as well as playing a significant role in other capabilities of interest – such as episodic memory, reflection, and reinforcement learning – by providing a primitive form of arithmetic directly within conditionals. As an example, consider the transformation in Figure 40. The key pattern here is Image(o:4 x:*x*/2+1 y:*y*), which operates on object 4 in a mental image, preserving its y dimension while shrinking its x dimension by a factor of 2 and offsetting it by 1 (so that the center of the object remains fixed). Although the pattern may appear to involve just addition and multiplication of individual numbers, the figure makes it clear that such a transformation actually operates on entire functions, with an offset shifting the whole function along its variable's dimension, and a coefficient expanding, contracting, or reflecting the function.

In the transformation just exemplified, both the coefficient and the offset were specified as constants. Yet they could, at least in principle, have been specified by variables. A variable in an offset implies that what is desired is the sum of two (random) variables, which in general implicates a *convolution*. Although convolutions have not yet been implemented in Sigma, when the offset variable only has a single nonzero value, it can simply be extracted and used like a constant. This has been implemented in Sigma, and used for example in reinforcement learning to add rewards to projected values (Section 6.2.5). Coefficients can at present only be specified as constants.

```
CONDITIONAL Scale-Half-Horizontal
   Conditions: (Image o:0 x:x y:y)
   Actions: (Image o:4 x:x/2+1 y:y)
```

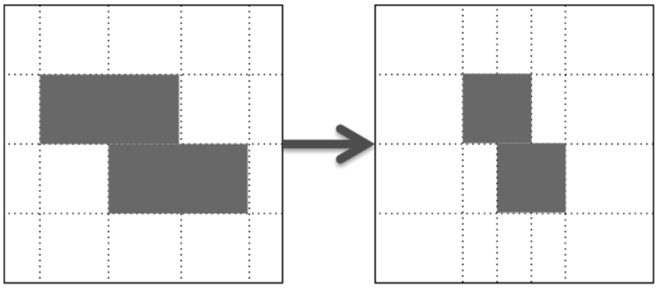


Figure 40: Scaling a Z tetromino by half, horizontally, in place.

When an affine transformation needs to be specified across multiple dimensions, translation and scaling can both be decomposed into individual components along each dimension.

However, rotation is trickier, as it inherently involves interactions across dimensions. Rotations by multiples of 90° are presently handled by swapping variables, but more about this will be said in Section 5.2.2.

4.5 Summary

The Sigma cognitive architecture has much in common with more traditional architectures, such as Soar and ACT-R, particularly Soar, with its appeal to problem spaces plus its cognitive cycle and the tri-level control structure it induces. The most significant extension provided by Sigma is the pervasive availability of *functions*, along with the bidirectional processing of the *conducts* that typically accompany them. This contrasts strongly with strictly symbolic approaches, and even approaches based on activation, in fully supporting *hybrid mixed processing*. In particular, this extension has been essential for enabling the representation and processing of distributions, images, and distributed vectors – and for the broadening of constant and variable tests to filters and affine transforms – within the elaboration phase of the cognitive cycle; and of flexible and pervasive selection, gradient-descent learning, appraisal and attention within the adaptation phase.

This core extension stems directly from the *graphical architecture hypothesis*, as functions and conducts are definitional in graphical models, even if the term *condact* is a new one. In the next section we will examine Sigma explicitly from the perspective of graphical models, including the way knowledge structures within the cognitive architecture compile down into such models in the graphical architecture.

5. Graphical Architecture

Below the cognitive architecture is the *graphical architecture*, with its core of *graphical models* (Section 5.1). The graphical architecture is essential to the simplicity and elegance at the waist of the Sigma hourglass and to the breadth and efficiency of the hourglass's top half. It is also what is implemented by the bottom half of the hourglass. As shown in the lower portion of Figure 41, processing within the graphical architecture – that is, the *graphical cycle* – is structured into two major phases, one concerned with solving graphs and the other with modifying them.

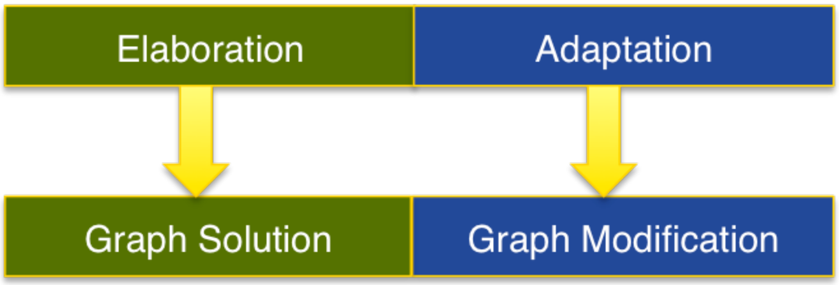


Figure 41: Mapping the two major phases in the cognitive cycle to the corresponding phases in the graphical cycle.

As revealed by Figure 41 as a whole, what is particularly notable about these two graphical phases is that the elaboration and adaptation phases from the cognitive architecture directly map onto them. The cognitive language is translated into a graphical language by a compiler (Section 5.4) that enables elaboration – including memory access, reasoning and perceptual processing – to occur by solving graphs (Section 5.2). The compiler also enables adaptation – decisions, reflection, learning, and affective and attentive processing – to occur by modifying these graphs (Section 5.3). For the classification scenario, the naïve Bayes classifier in Figure 7 is solved, and the rule in Figures 16(b) and Figure 31(b) is fired, all within a single graph solution. The best concept is selected as the operator as part of graph modification.

Very little will be said here about the details of the graphical language that serves as the target for the cognitive compiler. However, a few aspects that are critical for understanding how diversity is enabled at higher layers are included: in particular, the hybrid mixed function representation at the heart of the graphical models plus the key *graphical idioms* and settable parameters that are exploited by the compiler. Graphical idioms are analogous to but distinct from cognitive idioms, being defined as they are one level down in the systems hierarchy. Their core purpose is to support structured compilation from the cognitive language to the graphical language. Although the compiler itself is not discussed until Section 5.4, other aspects of the mapping between the cognitive and graphical architectures will be detailed as appropriate in the prior sections. Readers interested in an introduction to the graphical architecture, but who are not interested in the way knowledge expressed in the cognitive architecture maps onto factor graphs in the graphical architecture, should feel free to skip Section 5.4 on the compiler.

5.1 Graphical Models

Graphical models in general provide an efficient means of computing with complex multivariate functions by decomposing them into products of simpler functions and then translating them into graphs. For example, the trivariate algebraic function $f(x,y,z) = y^2+yz+2yx+2xz$ can be decomposed into the product $(2x+y)(y+z)$ – or generically $f_1(x,y)f_2(y,z)$ – which can then be encoded as a factor graph, as in Figure 42. From such a graph, the marginals of the individual variables – that is, the function's values when all other variables are summarized out – can be computed efficiently, as can the function's global mode: for example, yielding maximum a posterior probability (MAP) estimation. The *meaning*, or *semantics*, of the graph is determined by the function it is to compute, with algorithms for solving such graphs necessarily respecting this meaning.

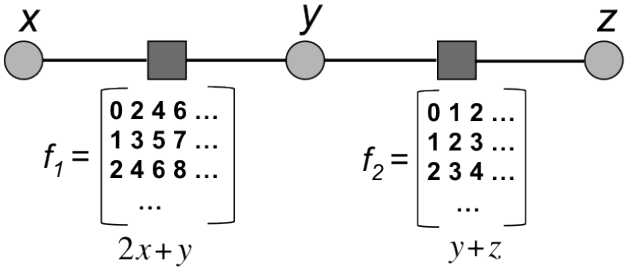


Figure 42: Factor graph for the algebraic function $f(x,y,z) = y^2+yz+2yx+2xz = (2x+y)(y+z) = f_1(x,y)f_2(y,z)$.

Major variants on graphical models include Bayesian and Markov networks, Markov and conditional random fields, and factor graphs. Bayesian networks are the most familiar within artificial intelligence and cognitive science. They provide an efficient means of computing with joint probability distributions, by decomposing them into products of prior and conditional probabilities, and then translating the results into directed acyclic graphs that have a node for each random variable and a directed link from one node to another if and only if the distribution of the second depends directly on the value of the first. Each node defines the conditional probability of its variable (the *child*) given those variables upon which it depends (its *parents*). The naïve Bayes network in Figure 7 is one such network. Figure 43 shows just the concept (*k*) from this classifier plus two of its features – number of legs (*l*) and color (*c*) – along with the equation for this reduced graph.

Markov networks and Markov/conditional random fields are similar to Bayesian networks, but they exploit undirected graphs that may be cyclic, along with *potential functions* that are defined over cliques of variable nodes. Both these directed and undirected forms of graphical models are typically used for probability distributions, but their precise expressibility varies, with some distributions not directly expressible in directed acyclic graphs and others not in undirected cyclic graphs.

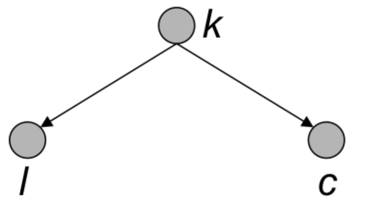


Figure 43: Bayesian network for the distribution $p(k,l,c) = p(k)p(l|k)p(c|k)$.

Factor graphs, as in Figure 42, are also undirected and cyclic, but they reify clique potentials as *factor nodes* in the graph itself, yielding bipartite graphs in which factor nodes connect to all of the variable nodes whose variables are used in their functions. Variable nodes in turn connect to all of the factor nodes that use them (Figure 42). Factor graphs were first developed in coding theory, where they underlie the remarkably effective performance of turbo codes, and are the most expressive form of graphical model known, subsuming both Bayesian and Markov networks (Kschischang et al., 2001). Figure 44, for example, shows how to encode the Bayesian network from Figure 43 as a factor graph. Factor graphs can be applied to arbitrary multivariate functions, not just to probabilistic functions, as seen in Figure 42 for a simple algebraic function. This generality was the motivation for basing Sigma on them.

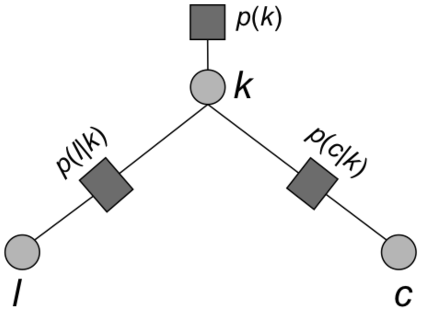


Figure 44: Factor graph for the Bayesian network in Figure 43.

As a factor graph, Sigma's graphical architecture can be characterized in terms of three interconnected memories, one each for factor nodes, variable nodes, and the undirected links that connect factor and variable nodes (Figure 45). Although these memories are shown as distinct boxes in the figure, in reality each consists of many individual elements that are intermingled, with individual factor and variable nodes being connected by individual links. All three of the cognitive memories – LTM, WM and PB – map into combinations of factor nodes, variable nodes and links at this level. Each memory in the cognitive architecture is thus a graphical idiom defined in terms of particular combinations of structures in the graphical architecture's memories.

The traditional way to specify a factor graph, or any form of graphical model, is via a single expression for the whole function that is then mapped into a graph that reflects its semantics when associated with an appropriate solution algorithm. In Sigma, factor graphs are created in a more piecemeal fashion. Via the compiler, fragments of the overall graph are built based on the individual types, functions, predicates and conditionals specified in the cognitive language (Section 5.4). The meaning of the overall factor graph is thus determined bottom-up from its nodes, their connectivity, and the functions embodied in the factor nodes.

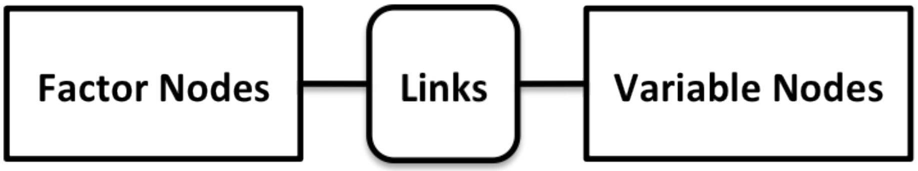


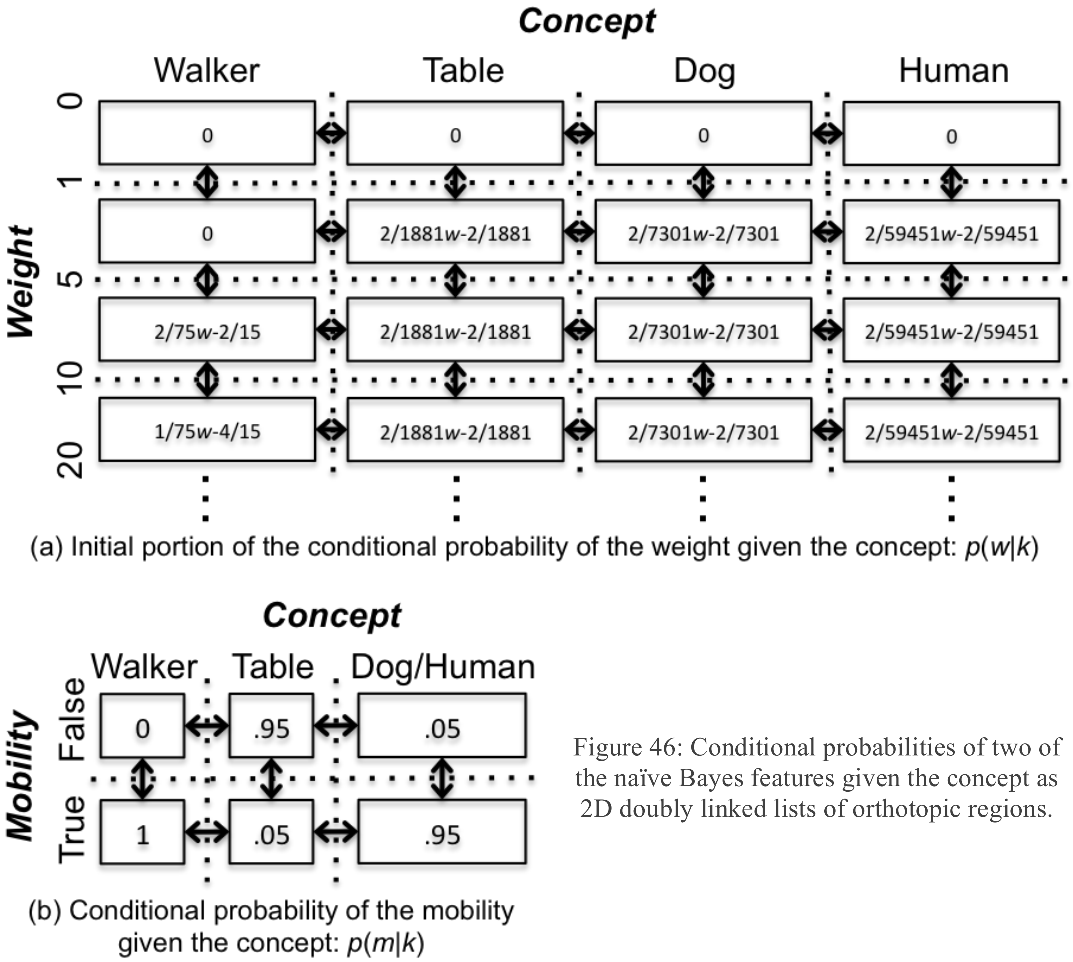
Figure 45: The three core memories in the graphical architecture.

The graphical language must therefore provide the ability to specify nodes and links, and to define factor functions. Nodes and links are straightforward data structures without much worthy of explicit note. However, the representation of factor functions is a critical determinant of the expressibility and efficiency of the resulting system. Some natural representations support only Boolean functions, ruling out the explicit encoding of uncertainty that is required for a mixed approach, whereas others support only either discrete or continuous functions, ruling out hybrids. Likewise, function representations may vary significantly in terms of the efficiency with which they can be processed during graph solution and modification. The remainder of this section first provides more details on the function representation used in Sigma and then explains several key ways that Sigma's graphical architecture diverges from standard factor graphs.

5.1.1 FUNCTION REPRESENTATION

During Sigma's prehistory, an approach to function representation was explored based on *exptrees* (Rosenbloom, 2011a), an *n*-dimensional generalization of quad/octrees in which dimensions were bisected as necessary until regions with constant values were reached. Later this was structurally simplified to an *n*-dimensional doubly linked list of orthotopic regions, but now with the possibility of linear functions in regions. A dimension-spanning *slice* would exist in such a structure whenever any pair of regions across it had functions that differed by more than a small value of *epsilon*. Slices were automatically removed when no longer needed. As with exptrees, these data structures would thus automatically adapt in size, being as small as one region for a uniform function, or as large as thousands or millions of regions when there was much detailed structure to represent. Figure 46 shows two examples, for the conditional probabilities of the weight and mobility given the concept for the naïve Bayes classifier in Figure 7. Only the initial portion of the weight function is shown, but it is enough to see how the overall function is piecewise linear. The mobility function is piecewise constant, as in Figure 32, but here it is shown how slices can be eliminated – in this case the one between `dog` and `human` – when they are not needed to distinguish the functions on either side of the slice.

This was the standard representation used throughout most of Sigma's history. It was particularly efficient when functions were to be destructively modified, such as when a portion of a working memory function is to be changed during the adaptation phase; however, it could not efficiently represent sparse functions nor was it particularly easy to process in the systematic manner required during graph solution. Consider first the question of sparse functions that are zero almost everywhere. The Eight Puzzle board, as shown in Figure 12, provides a simple example. There are nine planes here, one per tile, and only one non-zero *x*,*y* region per plane. Thus there are only nine non-zero regions out of eighty-one total regions. As discussed in Rosenbloom, Demski and Ustun (2015), an experimental version of Sigma has been explored with a sparse representation, in which (1) only non-default – typically non-zero – regions were explicitly represented and (2) region boundaries along a dimension did not need to align into slices. This approach did speed up processing over large sparse functions, but there was a trade off, in that region indexing was more complex so that processing was slower for large dense functions.



Weight	Walker	Table	Dog	Human
0–1	0	0	0	0
1–5	0	2/1881*w*-2/1881	2/7301*w*-2/7301	2/59451*w*-2/59451
5–10	2/75*w*-2/15	2/1881*w*-2/1881	2/7301*w*-2/7301	2/59451*w*-2/59451
10–20	1/75*w*-4/15	2/1881*w*-2/1881	2/7301*w*-2/7301	2/59451*w*-2/59451

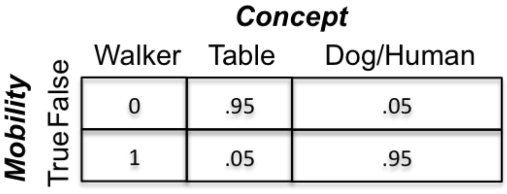
(a) Initial portion of the conditional probability of the weight given the concept: $p(w|k)$

Mobility	Walker	Table	Dog/Human
False	0	.95	.05
True	1	.05	.95

(b) Conditional probability of the mobility given the concept: $p(m|k)$

Figure 46: Conditional probabilities of two of the naïve Bayes features given the concept as 2D doubly linked lists of orthotopic regions.

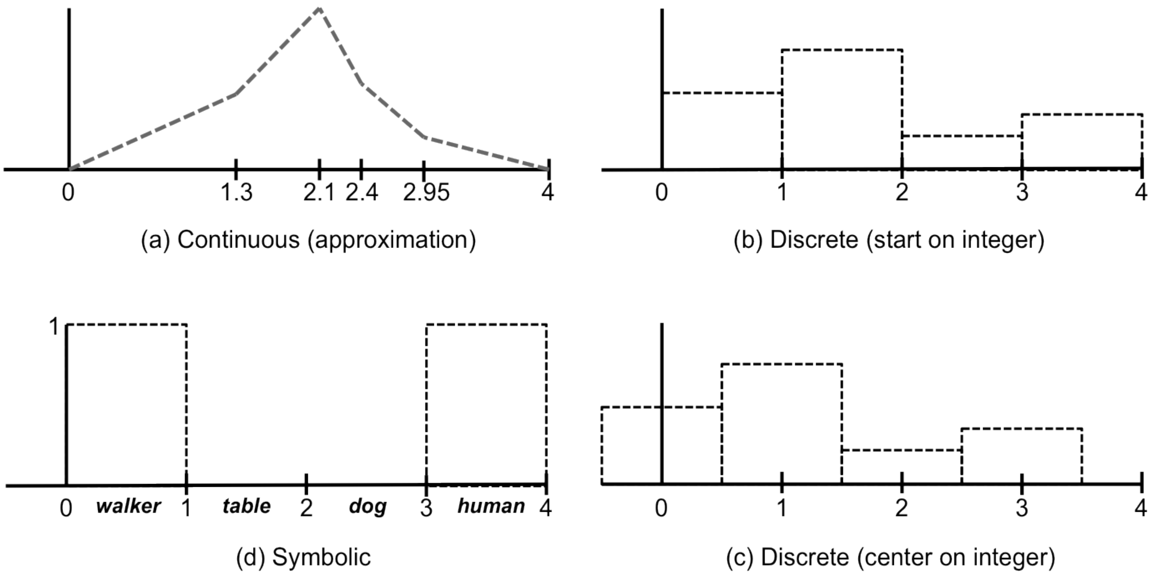
In the most recent versions of Sigma, functions are structured as *n*-dimensional arrays of orthotopic piecewise-linear regions (Figure 47). This sacrifices the ease of modification provided by the list-based representation in return for a more compact representation that is easier to traverse. Destructive function modification turned out to be the exception rather than the norm, so not too much had to be sacrificed by always requiring copying during function modification. In addition, a more efficient form of sparse function product was developed for use with these region arrays that yields some of the computational gain that was achieved by the sparse representation (Rosenbloom, Demski and Ustun, 2015). This representation can be viewed as a generalization of a pixel (or voxel) array, where (1) there can be any number of dimensions; (2) the pixels can vary in size along each dimension; and (3) the functions can be linear rather than just constant within each region. As with the earlier list-based representation, slices are automatically removed when no pairs of regions across them differ by more than epsilon. Figure 47 shows in this manner the conditional probability of the mobility given the concept from Figure 46(b).



Mobility	Walker	Table	Dog/Human
False	0	.95	.05
True	1	.05	.95

Figure 47: Conditional probability of the weight given the concept from Figure 46(b) as a 2D array of piecewise-constant orthotopic regions.

Although this representation is continuous at its core, it is general enough to support both hybrid and mixed processing. It can approximate arbitrary continuous functions as closely as desired when small enough regions are used (Figure 48(a)); yet, both discrete and symbolic functions can be represented via appropriate idioms that limit how these continuous functions are used, much in the manner that digital circuits arise from restrictions on a continuous underlying electrical substrate. Discrete functions can be represented via regions of unit length, with functions over these regions limited to constant values (Figure 48(b)). Such regions can potentially either start at integral values (Figure 48(b)), or be centered on them (Figure 48(c)). The default in Sigma is the former, which is conceptually simpler. However, the latter is also available as an alternative that enables the expected value of a constant function of `1` over such a region to equal its corresponding integer.



(a) Continuous (approximation)

(b) Discrete (start on integer)

(d) Symbolic

(c) Discrete (center on integer)

Figure 48: Idioms for representing continuous, discrete and symbolic functions.

For discrete variables, a half-open interval such as `[0,5)` comprises the integers from `0` up to one less than the maximum specified, yielding a largest value of `4` here. Purely symbolic functions are handled like discrete ones, except that the constant region functions are further limited to Boolean: `0/1` for `false/true` (Figure 48(d)). Symbolic variables are accompanied by symbol tables that are defined by the compiler, such as that `[walker table dog human]` maps onto `[0,5)` for concept names. But these are for human use only, and have no direct effect on how the graphical architecture either solves or modifies factor graphs.

Although a pure symbolic function is Boolean, with functional values limited to `0` or `1`, by leveraging the mixed aspect of the representation it is possible to define a non-Boolean function over a symbolic domain, to represent for example uncertainty over a choice among symbolic values. Functions can also be hybrids if they are composed of multiple variables of different types, as in the Eight Puzzle example presented earlier (Figures 10-11).

5.1.2 EXTENSIONS TO FACTOR GRAPHS

Several extensions have been made in Sigma to the standard notion of factor graphs in service of supporting desirable aspects of the cognitive architecture (Rosenbloom et al., 2016). Some are merely special purpose optimizations that replace normal factor graph structures and processing while still respecting the overall semantics. Others have a less clear status with respect to factor graph semantics, and may ultimately need to be considered as yielding some form of *extended factor graph*. Ideally, the semantic impact of any such extension would be clear and well understood theoretically, but much of this remains for future work. To the extent that such an understanding does not exist, and until it does exist, some sacrifice of functional elegance must be the consequence. More specifically, some theoretical elegance in the formulation of the graphical models and some guarantees about their solution algorithm may be sacrificed, at least in the short run, to produce given our current state of knowledge the desired cognitive functionality in an elegant manner from the graphical architecture.

One straightforward extension of basic factor graphs is to allow, in a manner similar to *junction trees*, each variable node to correspond to one or more function variables. This is critical within Sigma in solving the *binding confusion* problem, of determining which values go together across the marginals of multiple variables (Tambe and Rosenbloom, 1994). For example, this makes it possible to keep straight that there is a circle that is blue and a square that is red, rather than only knowing that there is a square and a circle and that one object is blue and the other red. In such a case, a single variable node may have variables for both `shape` and `color`. Logically this is the same as having a single new variable `shape×color` that is defined as the cross product of the two ground variables.

One other extension that is purely an optimization is the addition of specialized factor nodes for affine transforms. Factor functions in general can perform affine transformations via *delta functions* that, for continuous functions, are effectively of zero width and infinite height, with an integral of `1`. Figure 49, for example, shows a 1D delta function, and Figure 50 shows the 2D footprint of a shifted, angled, delta function that translates variable *x* up by one. Although factor functions are in principle capable of representing delta functions, they are not easily represented, or even approximated, via piecewise linear functions. So specially optimized nodes have been added in their stead, while maintaining the existing semantics of the factor graph.

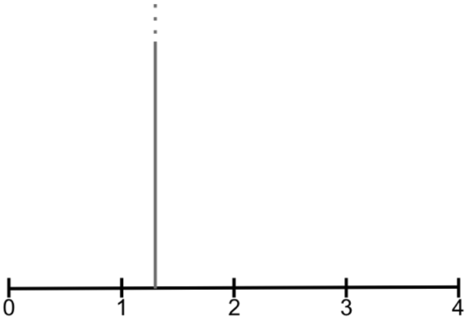


Figure 49: 1D delta function.

The most significant extension that involves ambiguous semantics is the ability to limit the direction of influence along a link. Along a normal link in a factor graph, the variable node influences the factor node and vice versa. However, in service of implementing conditions and actions, the option has been added to selectively cut off either direction of influence. It is important to note that this is not the same as the directedness found in Bayesian networks; the latter concerns the direction of variable dependencies in conditional probability distributions rather than the direction in which influence

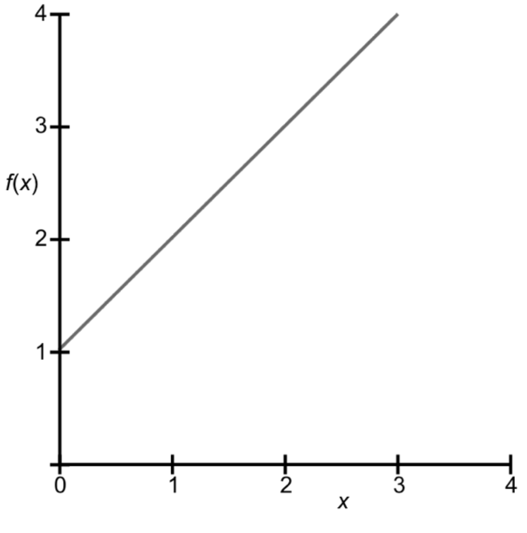


Figure 50: Footprint of 2D delta function for shifting variable *x* up by one.

spreads in the network, and so maps in Sigma onto choices of unique and universal variables in functions.

To the extent that the influences eliminated are irrelevant to the final marginals or MAP estimation, there is an aspect of this approach that amounts purely to optimization. However, it cannot yet be guaranteed that this yields well-defined factor graphs in the traditional sense. Instead, what currently exists is a non-semantic extension of the notion of a factor graph, plus an associated solution algorithm, as a useful and general processing mechanism.

A final class of extensions worth mentioning are special purpose factor nodes, for filters, action combination, and negations, that may yield an appropriate influence in one direction but not necessarily in the other. Action combination and negation also introduce the additional complication that, rather than just computing functions on the domains of the variables as is normal for factor nodes, they are computing functions on the distributions over the domains of the variables. Because these nodes appear close to the periphery of the graph, as will be explained in Section 5.4, they can almost be viewed as forms of pre- or post-processing that are outside of the factor graph. Thus, in one sense, they can be conceived of as not being part of the normal solution process for the factor graph. However, by incorporating them into the graph, a generalized notion of graph solution can provide a more uniform means of implementing these necessary forms of processing within the graphical/cognitive cycle.

5.2 Graph Solution

Solving a factor graph requires applying one of the many inference algorithms available for computing the values of variables in graphical models. Such a solution typically involves providing *evidence* for some of the variables – for example, by fixing their values via peripheral factor nodes – and then either computing the marginal distributions over the other variables individually or the modal value jointly over all of them. Given that evidence is usually supplied for only some of the variables in the graph, and may even then only cover some regions of these variables, a default value is required for unspecified variable regions. Typically, in Sigma, this is based on the open-world versus closed-world nature of the relevant predicate in the cognitive architecture.

The following subsections introduce the *sum(mary) product algorithm* (Kschischang et al., 2001) (also known as the *belief propagation algorithm* [Pearl, 1988]), as it has been adapted to solve graphs in Sigma, before delving further into three more specialized topics of importance for graph solution in Sigma: (1) special purpose implementations of factor nodes that do not follow standard summary product computations; (2) closure of the function representation over the operations performed by the summary product algorithm; and (3) several important optimizations that have been incorporated within graph solution, including initial work towards parallelizing message passing.

5.2.1 SUM(MARY) PRODUCT ALGORITHM

The sum(mary) product algorithm provides a message passing approach to computing graph marginals and modes in Sigma. Messages – structured as piecewise linear functions – are sent in both directions along links, from factor nodes to their neighboring variable nodes and from variable nodes to their neighboring factor nodes (Figure 51). Irrespective of the message's direction, what it represents is always a distribution over the variable node's variables. Figure 52 shows how the summary product algorithm can compute the marginal on the concept (k) given evidence that the color (c) is `silver`, and no evidence – that is, a uniform distribution – on the number of legs (*l*), for the reduced naïve Bayes classifier in Figure 44. For presentation purposes, the figure has been simplified to show only the meaningful messages going upwards in the graph rather than the full set of bidirectional messages, and to display the messages in terms of vectors and arrays rather than via piecewise linear functions. The details of the processing illustrated in this figure will be explained as this section proceeds. In Sigma, there would also be perceptual factor nodes not shown here that would be connected to each of the three variable nodes. Instead, the perception for both legs and color is shown adjacent to the corresponding variable nodes.

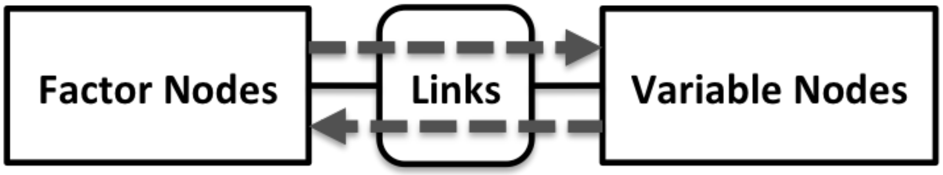


Figure 51: Graphical architecture with bidirectional messages passing over the links between factor and variable nodes

In general, the summary product algorithm begins by initializing all of the messages in the graph. In Sigma, all messages start on the first cycle with a constant value of 1, with two exceptions: messages arriving at action-combination nodes, which are initialized to 0 so that they don't saturate the outgoing messages, masking inputs that would otherwise provide meaningful content; and messages from *assumption nodes* – that is, factor nodes whose output can't change during the solution to the graph – which are initialized to the function stored in the node. In Figure 52, what is shown is not the initial messages, but the "meaningful" ones that are sent before the end of processing.

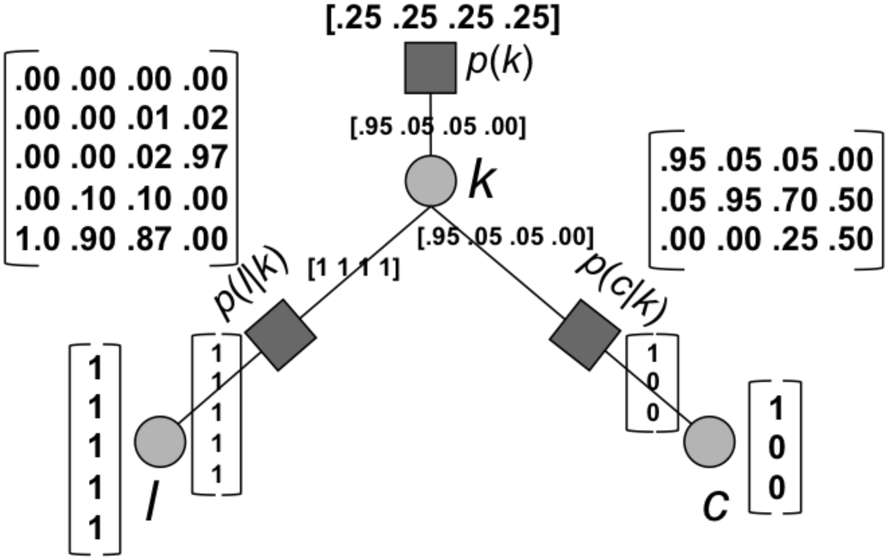


Figure 52: Message passing via the summary product algorithm for the marginal on the concept (*k*) in the factor graph of Figure 44.

Assumption nodes derive their name from an analogy to assumption-based truth maintenance systems (ATMS) (de Kleer, 1986), and include (1) evidence factors, which specify the latched contents of working memory in the cognitive architecture; (2) factors for predicate and conditional functions in long-term memory; (3) perceptual factors; and (4) discount factors, which enable the currently latched contents of working memory to become the default values for selection during the next cycle (as discussed further in Section 5.4). For brevity, the functions in these nodes can be referred to as *assumptions*. In Figure 52, all three of the factor nodes embody assumptions for predicate functions. The unshown perceptual factor nodes also embody assumptions. Figure 53 extends Figure 51 to explicitly mark a portion of the graphical architecture's factor-node memory as comprising assumption nodes.

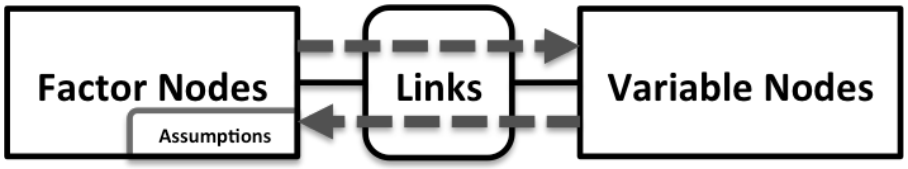


Figure 53: Graphical architecture with assumption nodes explicitly marked off within the factor-node memory.

Once the messages are initialized, whenever a node receives a new incoming message along a link, new outgoing messages can be generated for the node's other links (although messages are now generated in a lazy manner so that a message isn't actually generated until it is time to send it [Rosenbloom, Demski and Ustun, 2015]). Normally, new messages are generated for all of these other links; however, to implement the option of shutting off directional influence, each *link direction* – a combination of a link and a direction of the message passing along the link – can be individually deactivated so that messages are not passed in that direction along the link. Each link can therefore pass messages bidirectionally, or in either individual direction and not in the other, or in no direction. Bidirectional message passing captures the full semantics of factor graphs, as used in condacts and in Figure 52, while passing messages in neither direction deactivates the link without physically removing it from the graph. Unidirectional message passing is used for both conditions and actions, where working memory should either influence processing in the graph or be influenced by it, but not both.

At a variable node, the new outgoing message on each link is the pointwise product of the incoming messages on all of the other links. This is the *product* aspect of the summary product algorithm, which can be thought of in general as a means of conjunctively combining the evidence/constraints provided by the messages on the incoming links. This can be seen most clearly in the simple Boolean case, where `0` is mapped to false and `1` to true. Here pointwise product effectively *ands* the messages, as for example when the Boolean vectors `[0 1 0 1]` and `[1 0 0 1]` are multiplied to yield `[0 0 0 1]`. More generally, the message from variable node *k* to factor node *p*(*k*) in Figure 52 is computed by multiplying together the messages to *k* from the other two connected factor nodes, *p*(*l*|*k*) and *p*(*c*|*k*), to yield `[1 1 1 1]*[.95 .05 .05 .00] = [.95 .05 .05 .00]`. When normalized, this yields a distribution over the concept of `[.90 .05 .05 0]`.

Except for those with special purpose implementations, factor nodes compute the same pointwise product, but the node's own function is also included in the product. For example, in Figure 52 factor node *p*(*c*|*k*) computes a message for variable node *k* by multiplying the message from variable node *c* – `[1 0 0]`, which indicates that the color is definitely `silver` – and the function in node *p*(*c*|*k*) to leave the top row of the function unchanged while the two rows below it, for `brown` and `white`, become `0`. This isn't the final message, as it still needs to be summarized, so it isn't shown in the figure. The process is the same for computing the product for the message from factor node *p*(*l*|*k*) to variable node *k*, but here a product with a uniform input message – all 1 – just yields the original function.

Unlike at variable nodes, where the outgoing message is simply the message product, further processing is required to compute a normal factor node's outgoing message. Because the product here includes all of the node's variables, not just those corresponding to the variable node on the outgoing link, all other variables must be summarized out via a form of disjunctive combination. This is where the *summary* aspect of the summary product algorithm becomes relevant. For computing marginals in factor graphs, sum is normally used for discrete functions and integral for continuous functions. But, since discrete functions in Sigma are just idioms over continuous functions, integration is always used – the unit widths of discrete regions ensure that integrating over them yields the same result as simply adding their constant values. In Figure 52 the product computed above at factor node $p(c|k)$ thus must have the color integrated out to yield the message `[.95 .05 .05 0]` to variable node k. This is just the top row of the function, as any message with a single `1` and the rest `0` effectively acts as an index into a function when the two are combined in this way via the summary product algorithm. For the message from node $p(l|k)$, integrating out the legs leaves a uniform, all 1, message – illustrating how a feature with no evidence, and thus a uniform distribution, has no impact on the choice of concept. For computing functional modes rather than marginals, such as for MAP estimation, maximum is used instead of integration. Maximum effectively computes a pure *or* when the values are Boolean.

In general, the summary product algorithm can be used for any pair of a *combination* operation (such as product) and a *summarization* operation (such as sum/integral or max) that define a *commutative semi-ring*, where both operations are associative and commutative and have identity elements, and the distributive law exists (Kschischang et al., 2001). Sum/integral-product and max-product are common variations, but others are also possible. Sigma always uses product for message combination, except when it is done at special purpose factor nodes such as are used for action combination. Integral is used by default when summarizing over unique variables, although this can be changed to maximum when, for example, the mode of the graph, or of some part of it, is to be computed. Maximum is always used when summarizing over universal variables. Sigma thus employs a mixture of the sum/integral-product and max-product algorithms. When both unique and universal variables are to be summarized out at the same node, the unique variables are summarized out first.

5.2.2 SPECIAL PURPOSE IMPLEMENTATIONS OF FACTOR NODES

Most factor nodes with special purpose implementations still utilize the product of the input messages, but may process it in ways that don't merely involve multiplying it by a factor function, as with affine transforms, action combinations and negations. Or, factor nodes may do a multiplication but in a way that only works correctly for one link direction – as with filters – thus necessarily taking advantage of the fact that the other direction is deactivated. Action-combination nodes explicitly use operations other than product in combining their inputs.

For example, what affine factor nodes do is dependent on a set of (optional) parameters specifying the offset, the coefficient, and several other more minor aspects. The offset specifies a constant or variable whose value determines how much the function is to be translated to the right or left, depending on the sign of the offset. This amounts to simple addition of the offset to the locations of the region boundaries. Likewise, the boundary locations are multiplied by the coefficient to yield scaling – up or down depending on whether the coefficient is more or less than one – and to yield reflection when the coefficient is negative. Transformed functions may also need to be padded or cropped if the transformation causes them to be too small or large, respectively, for the transformed variable's type. Padding is by `1` for open-world predicates and `0` for closed-world ones unless these default values are overridden.

One other form of processing that can occur at affine factor nodes is variable swapping, in support of limited forms of object rotation (and several other functionalities that will be discussed in Section 5.4). Consider the tetromino in the left half of Figure 40, with its continuous *x* and *y* dimensions. If these two variables are swapped – in combination with a reflection to return it to the right sense and an offset to reposition it to where it started – then the result is a rotation by 90°, as shown in Figure 54. This form of variable swap, along with accompanying reflections and offsets as necessary, can yield rotations at any multiple of 90°. Rotation at arbitrary angles, however, remains for future work.

```
CONDITIONAL Rotate-90-Right
   Conditions: (Image o:0 x:x y:y)
   Actions: (Image o:3 x:4-y y:x)
```

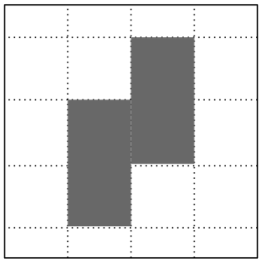


Figure 54: Rotation of Z tetromino by 90° in place.

5.2.3 CLOSURE OVER THE FUNCTION REPRESENTATION

Ostensibly the application of this version of the summary product algorithm to Sigma depends on its core function representation being closed under integration, maximization and product; that is, applying these operations to piecewise linear functions should yield new such functions. A piecewise linear function is integrated by computing the definite integral along the dimension of interest for each region, and summing the results across regions that share the same values for the other variables. Both definite integral and summation are closed over linear functions, so integration is closed over piecewise linear functions. Piecewise linear functions are, however, not closed under product. There is no issue if at least one of the operands is a constant function, but if both are linear the result is quadratic. To work around this, quadratic results are reapproximated to (piecewise) linear. Sigma currently does this in a simple region-by-region manner, by calculating the quadratic's average intercept and slope over the region's minimal, maximal and medial points. More sophisticated approximations, where individual quadratic regions are decomposed into multiple smaller regions, each with its own linear approximation, are possible, but have not so far proven necessary. Piecewise linear functions are closed under maximization, but in general this may also require decomposing individual regions into multiple smaller ones wherever two linear functions cross, since linear functions are not themselves closed under maximization in such situations. As with product, Sigma currently reapproximates each such region via a new linear function – in this case, averaging the two linear functions – rather than decomposing it at the crossover point.

5.2.4 OPTIMIZATION

Sigma's variant of the summary product algorithm provides an unusual form of a *lifted* inference algorithm (e.g., Singla and Domingos, 2008). Given first-order – that is, variablized – knowledge structures, a lifted algorithm reasons directly with these structures, whereas a *grounded* inference algorithm must create new instantiations of the graph structure for each combination of variable values and then reason over all of these ground graph structures. Sigma only reasons with variablized structures, but with the messages that are sent in the graph sliced as required to distinguish regions with distinct functions. Thus, a message may contain anything from a single region, implying that all combinations of variable values can be processed identically and together, up to a completely shattered representation of all possible variable values. This automatic lifting, to the extent allowed by the data, provides an important source of optimization in solving Sigma's graphs that naturally arises from its message representation.

In support of this, once a new output message along a link has been computed, unnecessary slices – that is, ones for which no spanning pair of regions differs by more than epsilon – are removed. In Figure 46(b) and 47, for example, the function has only three regions: one for `walker` at .95, one for both `table` and `dog` at .05, and one for `human` at 0. The same would be true in Figure 52 of the messages from nodes $p(c|k)$ and k if straight vectors weren't being shown. This does not in general change the meaning of the message, only its size in terms of number of regions. The one exception is for outgoing messages at predicate LTM assumption nodes when attention is enabled. In these cases, as described in Section 4.3.1, determining which slices aren't "needed" is based on the function resulting from the product of the scaled, exponentiated attention map and the outgoing message, with the results potentially leading to not only removal of "needed" slices from the outgoing message but also abstraction, via averaging, of regions across these slices. The implementation of this abstraction process is based on an extension to the preexisting mechanism for removing unneeded slices.

One relatively obvious but still critical further optimization to the graph solution process is based on the fact that it only makes sense to send an outgoing message if it is significantly different from the prior message along that link. To enable this to be checked, *message memory* has been introduced into the graphical architecture for storing the last message sent along each link direction, with summary product using this memory as an intermediate staging point in message passing (Figure 55).[5] The new message is compared to the one stored in message memory by determining if all corresponding parameters in all corresponding regions of the two messages are within epsilon of each other. For functional values, Sigma uses a relative epsilon, proportional to the magnitude of the numbers being compared, whereas it uses an absolute epsilon for all other comparisons. As with the other graphical-architecture memories, message memory is actually a collection of distributed structures, one associated with each link direction.

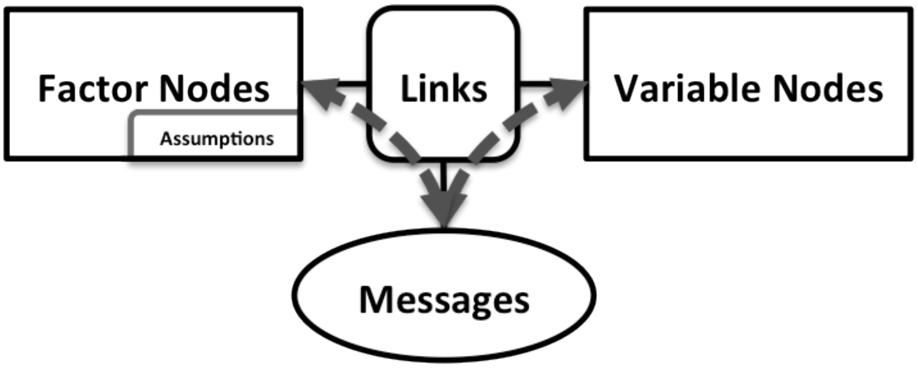


Figure 55: Graphical architecture with the addition of message memory.

Message memory also enables another optimization – message reuse across cognitive cycles – that is analogous to state saving/reuse across cycles of production match and firing (Forgy, 1982). The approach to message reuse currently taken in Sigma is based on Rosenbloom (2012c), where only messages dependent on changed assumptions are reinitialized. But because the recursive algorithm described for dependency analysis in that article does not scale well for graphs with many bidirectional links, it has since been replaced with a dynamic programming approach that is much more efficient on large graphs. For example, in a word sense disambiguation task in which each word in the vocabulary led to a distinct set of nodes in the

[5] In Rosenbloom (2011c), message memories were referred to as *link memories*, but that name is now used for the memory of links, as it seems more appropriate to name memories based on what they store rather than on their location.

graph, with many bidirectional links per node, the gain in speed was roughly proportional to the size of the vocabulary; for example, a speedup factor of 100 with 100 words and 1000 with 1000 words.

Sigma currently can only leverage a single thread of processing during graph solution, so message passing is serial, with just one message passed at a time (also known as *asynchronous belief propagation*). One consequence of this is that the order in which the messages are passed can be a major determiner of how many must be sent before quiescence is reached. Details on how messages are ordered in non-loopy portions of the graph, based on how far each link direction is from the assumption nodes that feed it – that is, its *depth* in the graph – can be found in Rosenbloom (2012c). To help with message ordering within loops, this approach has since been extended to link directions where a simple notion of depth doesn't apply. Given the depths for the link directions that are not affected by loops, the *loop depth* of a link direction that is affected by loops is computed by finding all of its nearest preceding link directions that do have assigned depths, and then assigning a loop depth that is the maximum over these predecessors of their depth plus their distance in links from the link direction. Experiments in Sigma with computing Nash equilibria over single-stage simultaneous-move games such as the *Prisoner's Dilemma* (Section 6.1.3), which are highly loopy, have shown speed gains of 15-20% with this extension.

On a parallel processor, messages could potentially be sent simultaneously in both directions along all links. For exploratory purposes, a form of simulated parallelism has been implemented that updates all output messages from all nodes on every cycle (also known as *synchronous belief propagation*) (Rosenbloom, 2012c). This makes it possible to calculate how many cycles of parallel message processing would be needed to solve a graph on a parallel machine. A true parallel implementation, however, has been left for future work.

Whichever of these two approaches is taken to message ordering, the natural stopping criterion for the graph-solution phase is *quiescence*; that is, when no significantly different messages remain to be sent. This works whether a single interconnected factor graph is defined or there are an arbitrary number of unconnected ones. It is analogous to the use of quiescence in Soar as a stopping criterion for the elaboration phase, although here it is messages that are passed until quiescence, whereas in Soar it is rules that are fired until quiescence.

5.3 Graph Modification

Once the graph-solution phase is complete, the graph-modification phase begins. In one sense this phase is extremely limited, as all it can do at present is modify functions associated with a subset of the assumption nodes. However, these do correspond to modifications of WM functions, LTM functions and some PB functions (for internal input), and thus map cognitively onto state changes, (non-structural) learning, and proprioception (taking this as the general capability of sensing internal state). As shown in Figure 56, the input for these modifications can stem from two sources: messages and assumptions. For completeness, the figure also shows one additional two-headed arrow connecting to the assumption memory, but it actually corresponds to the input phase (for external input into PB functions) and the output phase (for external action from WM functions), rather than representing part of the modification phase. No more will be said about input or output here.

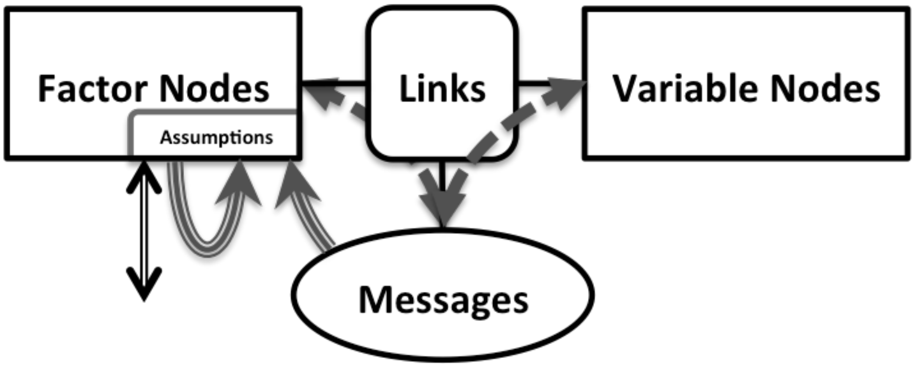


Figure 56: Graphical architecture with addition of graph modification (plus a two-headed arrow for input and output).

State changes in working memory, which occur only for closed-world predicates, such as `selected` (in operator decisions) and `board` (in the Eight Puzzle), involve latching WM functions by storing them into WM assumption nodes. There are three variants of this worth noting. The simplest variant is for unique predicates that maintain their full distribution, where the existing assumption is replaced by the message coming into the assumption node. A slightly more complex variant is for universal predicates, where the existing assumption is included as an implicit input to disjunctive action combination, implying that regions of the assumption not modified by explicit actions will remain as they are. The most complex variant is for unique predicates that make selections based on the distributions arriving via messages at the WM assumption nodes. When selection is to be of the best/argmax choice, if the existing choice is among the best proposed it is retained, otherwise one of the best is chosen at random. This default retention of an existing best choice is implemented in the graph itself by multiplying the existing assumption by a small *discount factor* (.01) before combining it disjunctively – effectively as a tiebreaker – with the actions.

Messages into WM assumption nodes in support of selection serve the function of *preferences* in Soar. The message memories in which these are stored thus also handle the functionality of *preference memory* in Soar. Soar's preference memory is generally considered an implementation detail rather than a first-class memory shown in architectural diagrams, but here it becomes part of a more general memory structure within the well-defined graphical architecture that sits below and implements the cognitive architecture.

Selection also raises a subtle yet conceptually thorny issue, in that it requires the graphical architecture to be aware of whether a variable is continuous or discrete – with symbolic variables treated the same as discrete ones – so that it can know how broad a region to use around the selected value: ε for continuous variables versus `1` for discrete variables. Thus, in this one sense, discrete variables are more than just idiomatic uses of continuous ones, albeit in graph modification rather than solution.

The two more complex forms of state changes in working memory were initially based on both messages and assumptions, thus necessitating special-purpose architectural code for combining them within the graph modification phase. But much of this processing has since been "regularized" in service of functional elegance by injecting the assumptions as messages into the graph in such a way that general-purpose graph solution can perform the combination (see Section 5.4 for more details on how this works). The result is that these changes are all now based solely on the messages arriving at the appropriate assumption nodes, yielding a simple model: messages coming out of WM assumption nodes drive solution while messages coming back into them drive modification.

Impasses, and the reflection they trigger, are also largely based on messages, in particular those arriving at the WM assumption node for the `Selected` predicate. This approach is sufficient for detecting *tie* and *none* impasses, but detecting *no-change* impasses also requires examining the existing assumption to determine which operator is already selected. Changes to the current impasse structure are then based on modifying the WM assumption for the `Impasse` predicate and extending or contracting the hierarchy of metalevel states.

Learning can occur at any LTM assumption node – such as node $p(k)$ in Figure 52 – based on the node's existing LTM assumption and the message arriving at the node. Leveraging the work in Russell et al. (1995) on learning in Bayesian networks, the incoming message turns out to provide the gradient needed by Sigma's gradient-descent learning algorithm (Section 4.2.4). Thus there is a symmetry here akin to the one just mentioned for WM, in which messages out of LTM assumption nodes drive solution while the messages coming back into them drive modification. The main difference between the two cases is the nature of the modification algorithm.

An earlier version of Sigma embodied an additional learning algorithm, for episodic knowledge, that was less local than the gradient descent algorithm in how it operated. However, later work showed that this special-purpose learning algorithm could be eliminated by deconstructing the requisite learning functionality in terms of gradient descent in combination with the appropriate conditionals to structure the graph (Rosenbloom, 2014).

Sigma uses proprioception to yield several appraisal variables (Section 4.2.3) based on a combination of messages and assumptions. Desirability is based on a goal WM assumption plus either a state WM assumption (for closed-world predicates) or the *variable posterior* – that is, the product of the incoming messages – at a goal WM variable node (for open-world predicates). Expectedness is based on the LTM assumptions before and after learning, which represent, respectively, the prior – or model – and the posterior distribution. Attention, which can be thought of as a higher-order or derived appraisal, is based on a combination of the perceptual assumptions for desirability and expectedness.

5.4 Compiler

The main task for Sigma's compiler is to convert predicate and conditional statements, along with their functions, from the cognitive language into fragments of factor graphs in the graphical language. The compiler also converts types into data structures that guide how functions are compiled and creates symbol tables for symbolic types that are usable during output to the human user, but little needs to be said about these aspects. The following subsections explain predicate compilation (Section 5.4.1) and conditional compilation (Section 5.4.2), and then illustrate these with examples of the compilation of a naïve Bayes classifier and a rule (Section 5.4.3).

5.4.1 PREDICATE COMPILATION

Predicate compilation is fairly straightforward, although a few subtleties result from specific characteristics of the predicates. The core of this process is the creation of a *working memory variable node* (WMVN) to which perceptual-buffer, working-memory, and long-term-memory assumption nodes (hereafter abbreviated as PBFN, WMFN and LTMFN, respectively) can be attached (Figure 57(a)). There is a PBFN if the predicate is perceptual, a WMFN if it is closed-world, and an LTMFN if it is memorial. In the classification scenario, there is a PBFN for the concept and each of the features in the naïve Bayes classifier, a WMFN for the `Selected` predicate in the rule (Figures 16(b) & 31(b)), and an LTMFN for each prior and conditional probability function in the classifier.

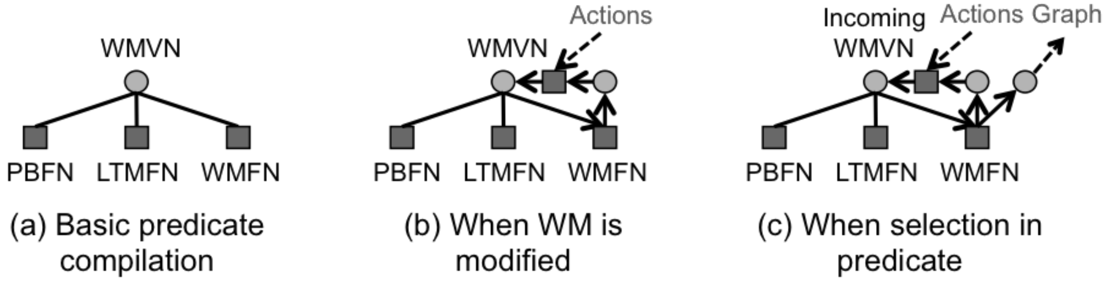


Figure 57: Schematic factor graphs for predicate compilation, assuming predicate is closed-world, perceptual and memorial.

There is a unidirectional link from a PBFN to its corresponding WMVN, since cognitive processing shouldn't impact input, whereas the connections with the other two types of assumption nodes are bidirectional in general. During graph solution (i.e., elaboration), results for a predicate are combined and inferences are chained through the WMVN, with the assumptions not changed until graph modification (i.e., adaptation), based on the quiescent messages arriving at their nodes.

If the predicate's entire distribution is not replaced each cycle – as, for example, with the Selected predicate – then the direct path from the WMFN to the WMVN is converted into an indirect path through a disjunctive action-combination factor node, to which actions can also be attached (Figure 57(b)). In effect, the current WM assumption is treated simply as another action. A discount factor node may also be included when appropriate in this subgraph. The outgoing link direction from the WMFN follows this new path, while the incoming link direction remains directly attached to the WMVN. The loop in this graph does not induce a loop during graph solution because the WM assumption does not change until the graph modification phase.

If the predicate also makes a selection over any of its variables – again as with the Selected predicate – then an extra WMVN is added to distinguish what is being proposed for selection during graph modification versus what is considered valid during graph solution (Figure 57(c)). The additional WMVN node is to support the use of the WM assumption during the elaboration phase, so an additional outgoing link is added from the WMFN to it.

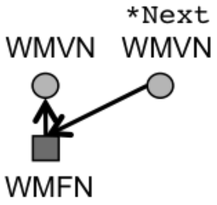


Figure 58: Graph structure for prediction.

When prediction mode is on, the graph is slightly different for closed-world predicates that include the state as one of their arguments (Figure 58). A corresponding open-world predicate with the suffix of *Next is created – so, for example, the closed-world Location predicate at the heart of SLAM is augmented by an open-world Location*Next predicate – with a unidirectional link from the open-world predicate's WMVN to the closed-world predicate's WMFN. This enables all of the changes for the predicate to be gathered in the *Next WMVN, somewhat like what is done for predicates that make selections, but here both the current and *Next functions are accessible during graph solution.

5.4.2 CONDITIONAL COMPILATION

Compiling conditionals is considerably more complex than compiling predicates, being based on a generalization of the Rete algorithm for rule match (Forgy, 1982) that has been extended beyond handling symbolic conditions to the full expressibility of Sigma's conditionals: in particular, also handling actions, condacts, functions, inversions, filters and affine transforms. This generalized form of Rete can be thought of as a large-scale graphical idiom that is leveraged by the compiler to convert conditionals into factor graphs.

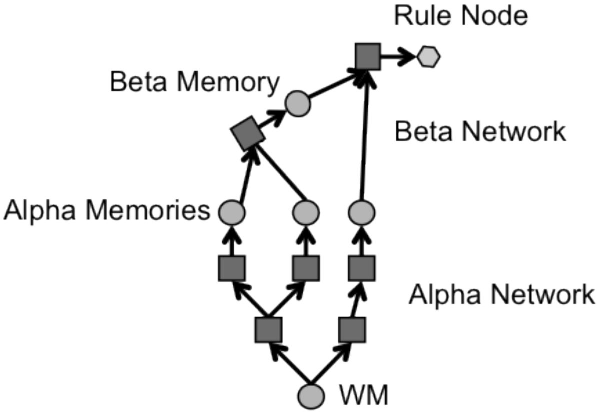


Figure 59: Schematic of Rete with processing and memory nodes (rather than factor and variable nodes).

Rete's match process is partitioned into an *alpha network* that handles individual conditions and a *beta network* that combines results from the alpha network into consistent matches across conditions (Figure 59). The alpha network is a *discrimination network* that branches on within-condition tests, such as for particular constants. The beta network is a linear *join network* that combines matches across conditions upon success of inter-condition tests, such as of variable equality and inequality. *Alpha memories* are inserted at the leaves of the discrimination network to maintain the state of the match for individual conditions. *Beta memories* are inserted after each join to maintain the state of the match across initial subsequences of conditions.

The match process in Rete is then driven by changes to working memory. When a *working memory element* (*WME*) is added or deleted, it is sorted through the alpha network until it is either filtered out or reaches the appropriate alpha memories. Changes to alpha memories stimulate the beta network, where they may combine with results from other alpha and beta memories to yield changes in later beta memories. As with changes to alpha memories, changes to beta memories may also stimulate processing in later regions of the network. Actions are triggered by changes arriving at the terminal nodes of the beta network – that is, at *rule nodes*. By maintaining intermediate results in alpha and beta memories, Rete achieves dynamic-programming-like efficiencies within a single match cycle; and, by maintaining these memories across cycles, match time becomes a function of the number of changes to working memory rather than its overall size. Rete also achieves a significant efficiency boost from sharing network structure (and memories) across comparable conditions and sequences of conditions.

If we break this down a bit more, when mapping Rete onto Sigma we can talk in terms of two smaller graphical idioms, one for alpha networks and the other for beta networks. In particular, conditionals compile into fragments of factor graphs structured according to these idioms, with the summary product algorithm providing the requisite message passing and computation. Because bidirectionality is the norm in graphical models, condacts rather than conditions become the most natural form of pattern supported, but both conditions and actions can be supported via the capability the graphical architecture provides for selectively deactivating link directions. For a condition, the link between the WMFN and the WMVN is deactivated towards the WMFN; and for an action, it is deactivated in the reverse direction. (For a condact based on an open-world predicate there is simply no WMFN.) In addition, all other link directions in the alpha and beta networks that pass messages towards working memory for conditions and away from it for actions become irrelevant, and so can be deactivated as a simple optimization that reduces the number of messages required for quiescence but doesn't change the meaning of the graph.

In consequence, all three forms of predicate patterns compile to comparable fragments of (extended) factor graphs, with the main difference being the direction of message passing within them. Figure 60 shows the full graph for a notional conditional with two conditions, one condact, and two actions (plus a conditional function). This figure will provide a useful reference throughout much of the following discussion.

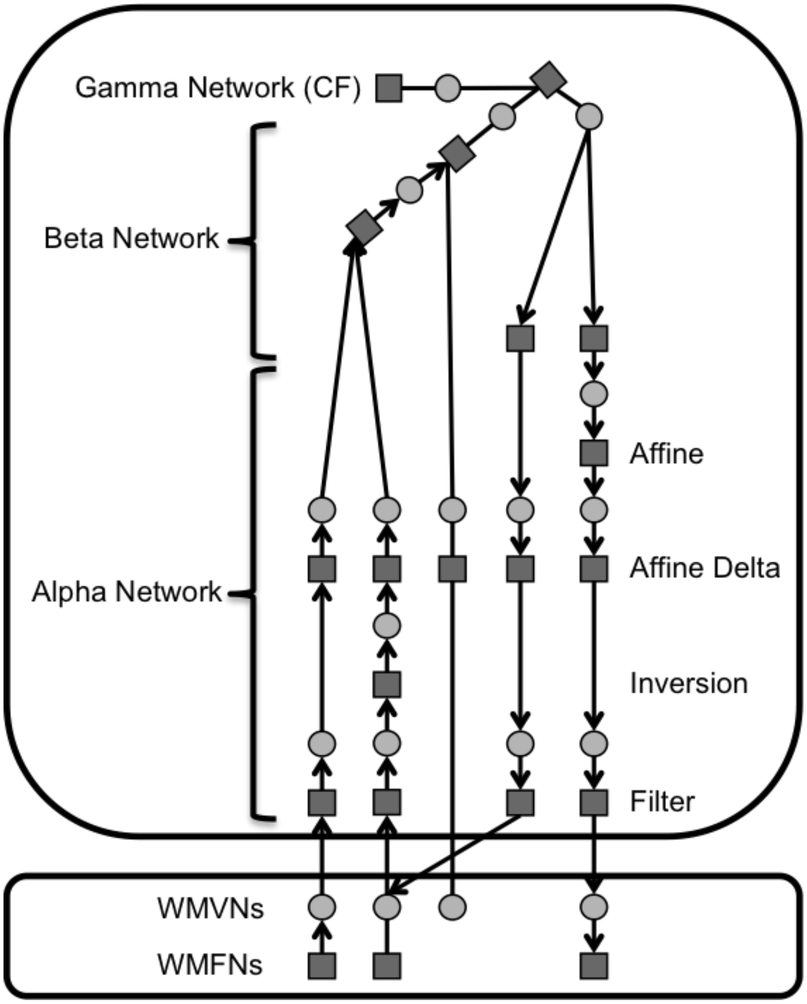


Figure 60: Graph for a conditional with two conditions (left two alpha networks, with arrows pointing away from WM), one condact (middle alpha network, with undirected links), two actions (right two alpha networks, with arrows pointing towards WM), and a function (CF). Four distinct predicates are involved, with the second one supporting both a condition and an action.

Because Sigma induces a distinct segment of working memory for each predicate, the first discrimination in the alpha network, on the name of the predicate, is implicit. Also, since we are showing a single conditional here whose conditions are based on different predicates, there is no further branching within the alpha network (sharing across conditionals is possible in Sigma through much of the alpha network, but not at present into the beta network). Therefore, each pattern instance here compiles to its own linear alpha subgraph that is anchored to the working memory for its predicate. The message memories for the terminal links of the condition subgraphs serve as alpha memories, maintaining the results of the subgraph processing in the direction of the beta network, while the message memories in the beta network serve as beta memories. Thus, just as Sigma's message memory subsumed Soar's preference memory (Section 5.3), it also subsumes the alpha and beta memories used in Soar's implementation of Rete (while also storing the last message along every link rather than just at particular points in the alpha and beta networks).

Within an alpha subgraph, Sigma handles all of the within-pattern processing for a single use of a predicate pattern. If there is a constant test, such as would be used for the pattern `Concept(object:`*`x`*` value:walker)`, a factor node for it is included in the alpha subgraph. Consider first what should happen if this pattern were to serve as a condition. A message should go forward that is nonzero only for objects that are of category `walker`, and which retains the functional value found in the corresponding region(s) of the incoming message. This is easily implemented in the factor node via a functional value of `1` where the `value` is `walker` and a value of `0` everywhere else. This is also then easily generalized to filters, as the same nodes must simply handle piecewise linear rather than merely piecewise constant functions.

This same approach straightforwardly extends to the use of constant tests and filters in actions as well. Figure 60, for example, shows filters, which may be constant tests or more general piecewise linear functions, in both the conditions and actions. The situation is somewhat more complicated for the use of filters in condacts, and with open-world predicates in general. Sigma is still in flux concerning whether the filter functions here should be 1 outside of the area of concern – corresponding to the open-world notion *unknown* – rather than 0. The standard at this point is for 0 to be used for all open-world patterns, except for condact processing towards working memory, as these "outside" regions in condact filters should not constrain the messages coming back into WM. However, this is likely to evolve in the future, with hopefully a less ad hoc solution still to be found.

If a pattern is negated, as we saw in Figure 37, its alpha network includes an inversion node, as in the second condition of Figure 60. Affine transforms, such as for the conditionals in Figures 40 and 54, may also be included, as in the second action in Figure 60. The variable swapping capability demonstrated in Figure 54 also turns out to subsume an additional form of processing within alpha subgraphs that previously required its own specially optimized nodes. Because a single set of working memory nodes is shared across all patterns for the corresponding predicate, the variables used in these nodes are not in general the same as those specified in the conditional patterns defined for that predicate. A factor node is thus needed – at least if these variables are used beyond the scope of the pattern under consideration – that makes the working memory variable equal to the pattern variable. This requires a delta function in general, which has already been mentioned as problematic. However, a specially optimized node can implement the requisite functionality simply by swapping the variables. Because affine nodes support variable swapping, the previously used special-purpose delta nodes could be eliminated. The compiler now automatically introduces more affine nodes – referred to as *affine delta nodes* – into the alpha subgraphs as needed for this kind of variable swapping, placing them between filter nodes and normal affine nodes, as shown in Figure 60.

Affine nodes also have one more trick up their sleeves. As will shortly be described, tests of variable equality across patterns are handled in the join nodes of the beta network, but variable equality within a pattern – as in `Test(a:`*`v`*` b:`*`v`*`)` – is tested within the alpha subgraph, at the affine delta nodes. Effectively, two distinct working memory variables must be mapped to a single pattern variable (*`v`*), yielding a `1` only for values common to both. This is implemented within affine nodes for conditions by diagonalizing along the two working-memory variables and using the result for the pattern variable. Reuse of variables within individual actions and condacts is not yet supported.

This completes the range of processing that can occur within a single alpha subgraph. So, the question now becomes how the subgraphs for patterns within a single conditional can be joined in the beta network. Conditional functions (CFs) also need to be compiled into fragments of factor graphs – forming what can be referred to in Rete-like terms as *gamma networks* – and joined to the conditional's beta network. In principle this all could occur via a single join node to which the alpha subgraphs and the gamma subgraph (should there be one) are directly attached. Alternatively, there could be a hierarchical join network that combines two subgraphs at a time or a linear join network that introduces one new alpha network at a time. A Rete-like linear join network is currently used in Sigma for conditions, condacts and functions, with conditions ordered before condacts and condacts ordered before the function (as in the first part of the beta network in Figure 60). Whenever a new variable is introduced by a pattern in the join network, the messages and join nodes from there until the pattern in which it is last used must include a dimension for that variable so that the constraint implied by its bindings is shared in all of the patterns containing it.

This accumulation of variables in intermediate factor nodes and messages bears a surface resemblance to the *junction tree* optimization that is commonly applied to loopy graphs. However, here the accumulation is necessary for correctness rather than merely being an optimization that enables the removal of loops from the graph. Still, as with junction trees, this accumulation of variables can significantly affect the efficiency of processing, with execution potentially combinatoric in the *tree width*; that is, in the maximum number of variables in a message or node (Rosenbloom, 2011a). On the plus side, this can still yield worst-case bounds better than those achievable by the standard form of Rete, which can be combinatoric in the number of (condition) patterns in a rule rather than in the number of variables whose values need to be simultaneously maintained.

To join subgraphs, each factor node in the beta network – that is, each *beta factor* – embodies a constant function of 1 over the union of the variables on its links. As incoming messages are multiplied times this function, the constraints from their variable bindings affect outgoing messages, with across-pattern variable-equality checks implicitly occurring when multiple uses of the same variable jointly constrain the same dimension. If an incoming variable isn't required in an outgoing message, the variable is summarized out from the outgoing message. This eliminates from messages along links in the linear join network the variables that aren't needed along those links because they are neither used in the variable nodes directly connected to the links nor required for comparisons in patterns further afield. It also eliminates variables not needed in outgoing messages to the alpha networks for actions or condacts, or in the gamma network for the function. The memories on the links within the beta network serve as beta memories, maintaining intermediate results, for example, for initial subsequences of conditions and condacts.

By subsuming the functionality of Rete's alpha and beta memories, Sigma's message memories yield Rete-like dynamic-programming gains within a single cycle; and, with the optimization discussed in Section 5.2.4 for message reuse across cycles, it maintains these gains across cycles. What is different, though, is the level of granularity of the messages that are being reused. In Rete, a message in the beta network is a *token* that comprises a single WME for each of the conditions joined, whereas in Sigma a message in the beta network effectively spans all possible tokens for the corresponding patterns. There is a potential for significant savings in Sigma when multiple tokens map into a single region of a piecewise linear function that can be processed as a unit, with these savings being (logically) infinite for continuous variables. However, if any part of such a message is altered, the entire message must be resent, possibly duplicating work for the unchanged portions. The possibility of messages representing only altered portions of functions remains an issue for the future.

Action patterns are consumers of variables introduced in conditions and condacts, rather than introducers of new variables that can further contribute to the combinatorics. It thus becomes feasible to attach all of a conditional's action subgraphs directly to the last join node in the linear sequence (see the latter part of the beta network in Figure 60).

In general, the subgraphs for different conditionals connect in working memory, at the WMVNs, implying that it is the WMVN nodes rather than the WMFN nodes that mediate result chaining across conditionals within the graph solution (or elaboration) phase. WMFNs can't mediate chaining for open-world predicates, as they do not even exist in such cases, and for closed-world predicates, changes to WMFNs aren't made until the graph modification phase. The subgraphs for condacts and conditions receive messages directly from the WMVNs, while action and condact subgraphs yield messages to them (Figure 61). In the latter case, actions and condacts are combined as discussed in Section 4.4.4. There are, however, cases in which there are many – thousands or millions of – condacts for the same predicate, where for computational reasons the large product at a lone WMVN is replaced with a hierarchy of nodes that compute the same result. The key to this is that the message memories in this expanded WMVN subgraph enable reusing partial products, and thus can dramatically reduce the cost of changing a single message in the overall product (Rosenbloom, Demski and Ustun, 2015).

5.4.3 EXAMPLES

As one example of a compiled factor graph, consider how the two-feature fragment – `Color`, which is perceived as `silver`, and `Mobile`, which is unperceived – of the naïve Bayes classifier from Figure 7 that is shown abstractly in Figures 44 and 52 compiles into an actual factor graph in Sigma. Figure 62 is a screen capture from Sigma's interactive factor-graph display. The relevant conditional for the `Mobile` attribute was shown in Figure 33, with the conditional probability distribution shown in Figure 32. The conditional for `Color` is similar, but the conditional for the prior distribution on the `Concept` is a bit simpler, comprising just two condacts, one that ties concepts to objects and the other for the concept's memorial predicate. The labels on the individual nodes are unreadable at this resolution, but the rectangles are factor nodes and the ellipses are variable nodes.

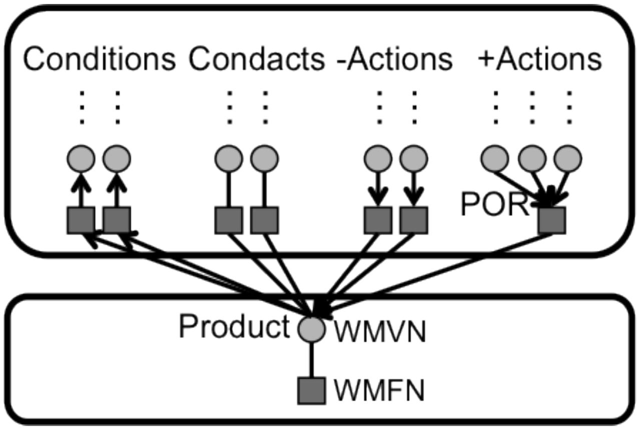


Figure 61: Connecting subgraphs in working memory for a single predicate.

At the top of the figure are the PBFN and WMVN for the `Concept`, with the node on the far right providing the LTMFN for the prior distribution on the `Concept`. The two longer chains ending with bidirectional links in this figure end at LTMFNs, whereas the two shorter chains ending with unidirectional links terminate with PBFNs. (None of the chains terminate in WMFNs because the predicates are all open-world here.) The variable nodes directly connected to the assumption nodes are the WMVN nodes. All of the non-assumption factor nodes connected to the various WMVN nodes – that is, those factor nodes towards the interior of the graph – are affine delta factors (ADFs) that change the names of the variables used in WM to those used in the patterns. The remaining factor nodes are all beta factors (BFs) that join together alpha networks. The two chains that join on the left connect the LTMFN for `Concept-`

Mobile with the PBFN for Mobile, while the similar structure in the middle does the same for Concept-Color and Color. These two subtrees connect at the Concept WMVN.

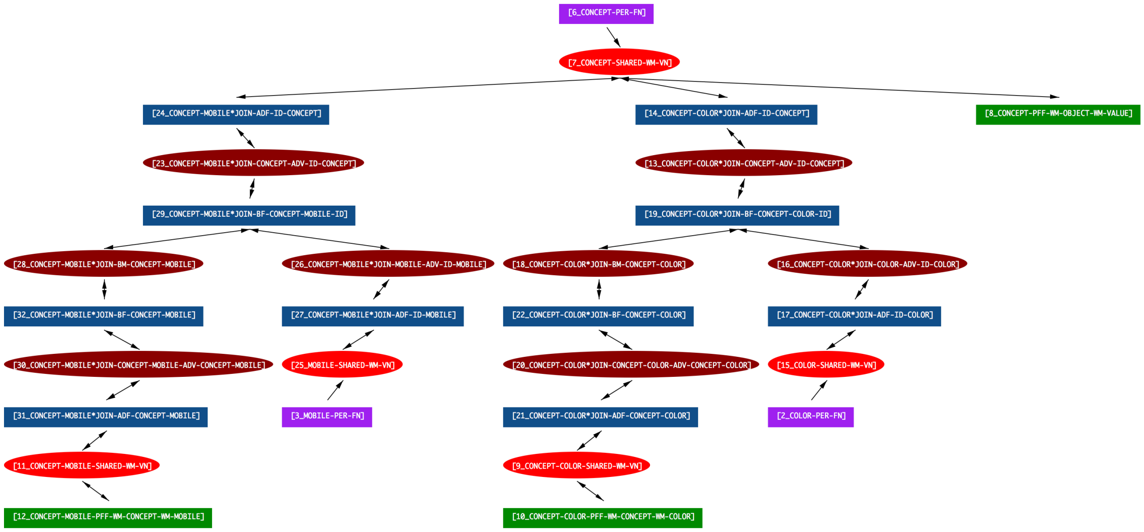

[6_CONCEPT-PER-FN]
[7_CONCEPT-SHARED-WM-VN]
[24_CONCEPT-MOBILE*JOIN-ADF-ID-CONCEPT]
[14_CONCEPT-COLOR*JOIN-ADF-ID-CONCEPT]
[8_CONCEPT-PFF-WM-OBJECT-WM-VALUE]
[23_CONCEPT-MOBILE*JOIN-CONCEPT-ADV-ID-CONCEPT]
[13_CONCEPT-COLOR*JOIN-CONCEPT-ADV-ID-CONCEPT]
[29_CONCEPT-MOBILE*JOIN-BF-CONCEPT-MOBILE-ID]
[19_CONCEPT-COLOR*JOIN-BF-CONCEPT-COLOR-ID]
[28_CONCEPT-MOBILE*JOIN-BM-CONCEPT-MOBILE]
[26_CONCEPT-MOBILE*JOIN-MOBILE-ADV-ID-MOBILE]
[18_CONCEPT-COLOR*JOIN-BM-CONCEPT-COLOR]
[16_CONCEPT-COLOR*JOIN-COLOR-ADV-ID-COLOR]
[32_CONCEPT-MOBILE*JOIN-BF-CONCEPT-MOBILE]
[27_CONCEPT-MOBILE*JOIN-ADF-ID-MOBILE]
[22_CONCEPT-COLOR*JOIN-BF-CONCEPT-COLOR]
[17_CONCEPT-COLOR*JOIN-ADF-ID-COLOR]
[30_CONCEPT-MOBILE*JOIN-CONCEPT-MOBILE-ADV-CONCEPT-MOBILE]
[25_MOBILE-SHARED-WM-VN]
[20_CONCEPT-COLOR*JOIN-CONCEPT-COLOR-ADV-CONCEPT-COLOR]
[15_COLOR-SHARED-WM-VN]
[31_CONCEPT-MOBILE*JOIN-ADF-CONCEPT-MOBILE]
[3_MOBILE-PER-FN]
[21_CONCEPT-COLOR*JOIN-ADF-CONCEPT-COLOR]
[2_COLOR-PER-FN]
[11_CONCEPT-MOBILE-SHARED-WM-VN]
[9_CONCEPT-COLOR-SHARED-WM-VN]
[12_CONCEPT-MOBILE-PFF-WM-CONCEPT-WM-MOBILE]
[10_CONCEPT-COLOR-PFF-WM-CONCEPT-WM-COLOR]

Figure 62: Sigma's interactive factor graph display for the reduced two-feature version of the naïve Bayes classifier from Figure 7 that is shown in Figures 44 and 52.

Given the perception of silver for the Color PBFN, and the conditional probability distribution in the Color LTMFN, the product of these two functions in the middle of the graph yields a distribution over the Concept once the Color is summarized out. This combines multiplicatively with the prior distribution on the Concept from the rightmost node to yield the posterior/marginal on the Concept. The messages from the Concept PBFN and the Mobile tree are both uniform because of lack of input for these predicates, so they have no influence on the posterior. However, given the posterior computed for the Concept, the message back to the Mobile tree combines with the LTMFN to generate a prediction for it in the WMVN that sits just above the PBFN. The messages arriving at the three LTMFNs drive learning from experience of the prior and conditional probability distributions. There is little to learn when only one feature is perceived and the Concept is also not seen, but there is no inherent difference between a learning trial and a test trial, so Sigma always tries to learn what it can given what information is available.

It is worth noting that this factor graph is considerably larger than the minimal factor graph shown in Figures 44 and 52 for this fragment of a naïve Bayes classifier in isolation because of the additional structures introduced by the compiler in implementing the generalized Rete idiom. (This is analogous to, but more extreme than, the expansion that occurs when converting a Bayesian network (Figure 43) to its equivalent in the more general formalism of factor graphs (Figure 44).) This expansion should yield a constant-factor slowdown over the minimal graph but not otherwise change what is computed. It is an open question whether additional compiler optimizations might eventually be able to reduce this extra overhead in situations where the full generality is not required.

As a second example of a compiled factor graph, consider the abstracted diagram in Figure 63 that shows how the transitive rule in Figures 16(a) and 31(a), which includes three patterns over a single universal, closed-world `Above` predicate, is compiled into a factor graph and then processed, given `Above(first:o1 second:o2)` and `Above(first:o2 second:o3)` initially in working memory, and thus a WMFN function with `1`s for just these two WMEs. In the first step a message is sent from the WMFN via a variable node to a *max* action-combination node whose other input is currently `0` (not shown).[6] Since `0` is the identity element for max over non-negative numbers, the result is the same as the message from the WMFN. This is then sent to the WMVN, and from there along the two condition paths. At the affine delta nodes, the variable names used in working memory (*`first`* and *`second`*) are converted to pattern variable names – *`a`* and *`b`*, and *`b`* and *`c`*, respectively – but the messages are not otherwise changed (ignoring for now the gray `1`s in the bottom left regions). The messages from the two conditions are then joined via product at a beta factor, with variable *`b`* summarized out to yield a message over variables *`a`* and *`c`*. This is then joined with the action, whose variable names are changed back to those used in working memory. When this message arrives at the max combination node it combines with the message from the WMFN to yield an updated message to the WMVN. From the WMVN this message is sent to the WMFN, replacing its previous function during graph modification with one that has `Above(first:o1 second:o3)` added to it. Simultaneously, the message reaching the WMVN is also passed back along the two condition paths to yield updated messages with `1`s in the bottom left regions (as shown in gray in the corners) rather than `0`s. When these new

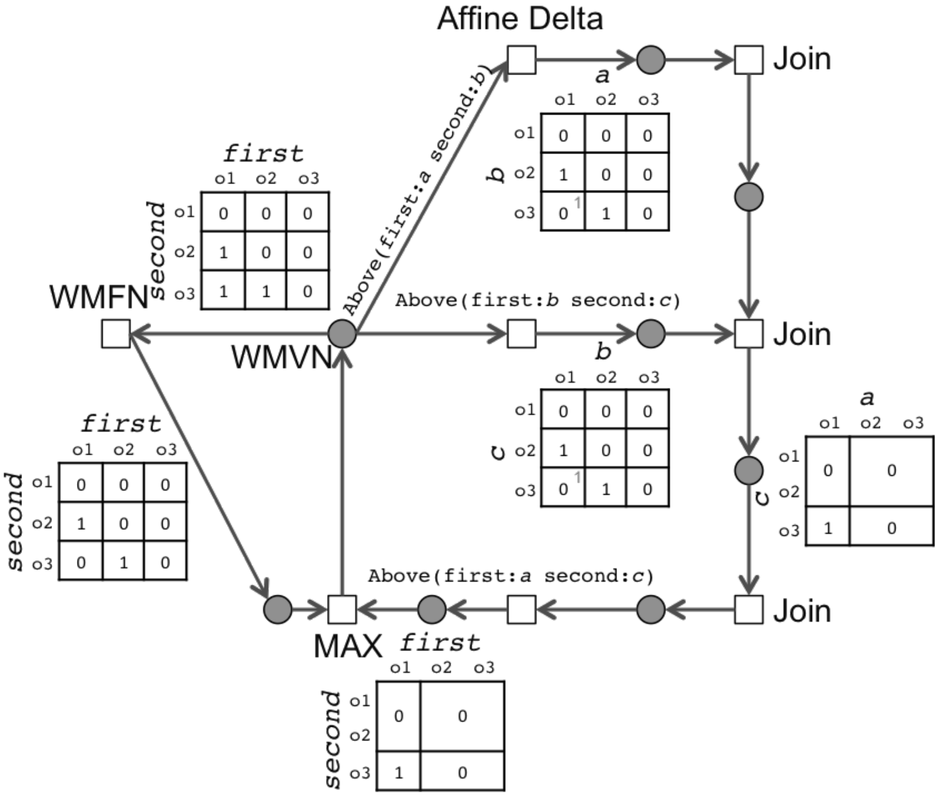


Figure 63: Abstracted factor graph and message passing for the transitive rule in Figure 15, given an initial working memory of `Above(first:o1 second:o2)` and `Above(first:o2 second:o3)`.

[6] As mentioned in Section 5.2.1, messages into action-combination nodes are initialized to 0 rather than 1.

messages are combined at the second join node, the output is unchanged from the previous message on the link, so processing reaches quiescence at that point.

The compiled factor graph for the rule in the classification scenario that is shown in Figures 16(b) and 31(b) follows a similar pattern, although it only has one condition rather than two, and the action is for a different predicate and involves a selection. This rule graph connects to the factor graph for the Bayesian network that is shown in reduced form in Figure 62 via the WMVN for the `Concept` predicate (the top-most variable node in the figure).

5.5 Summary

At its core, Sigma's graphical architecture is comprised of a factor graph that is solved by a mixture of the sum-product and max-product message-passing algorithms, and that is modified by altering its embedded assumptions; that is, by changing values in functions associated with PB, WM and LTM factor nodes. Several extensions have been made to the standard factor graph formalism and to the solution algorithm – some that continue to reflect classical factor graph semantics and others that, at least currently, do not – so that it could both serve as a better target of the cognitive compiler and generate solutions to the resulting graphs more efficiently. The compiler itself is built around a generalized version of the Rete algorithm, with numerous extensions to handle the increased expressivity of the cognitive language. Despite the warts, this generalized combination of factor graphs and Rete (as a graphical idiom that is used pervasively on top of the graphical base) provides a functionally elegant means of supporting the breadth of functionality provided in the cognitive architecture.

The next section shifts the focus up above the cognitive architecture, to examine how it yields an even wider range of intelligent capabilities as functionally elegant cognitive idioms.

6. Cognitive Idioms

A *cognitive idiom* is a form of stylized knowledge above the cognitive architecture that can be ascribed its own meaning. In a more conventional computing context, it is analogous to a software idiom, design pattern, library or service. In Sigma it can also be thought of as providing a form of *supraarchitectural capability* that should be broadly (re)usable in much the same way that an architectural capability is (Rosenbloom, 2015). For example, a *rule*, as in Figures 16(a) and 31(a), is a cognitive idiom in Sigma rather than an architecturally delineated knowledge structure. It is a conditional based on conditions and actions over universal closed-world predicates. Such rules behave much like those in Soar, firing in parallel during the elaboration phase, and chaining until quiescence is reached. A *Bayesian network* is also a cognitive idiom in Sigma, based on conditionals restricted to just condacts and a function; one unique variable – the child – per function; functions representing conditional (or prior) probability distributions; and no variable included as a child in more than one function. The naïve Bayes classifier(s) in Figures 7, 44, 52 and 62, and the reactive HMM in Figure 14, are simple examples of such networks.

These examples reveal that cognitive idioms can occur at multiple levels of granularity, here at the level of a single conditional or a combination of an arbitrary number of them. Many cognitive idioms can also be composed or decomposed into additional idioms, yielding a hierarchy of such idioms. For example, multiple rules can be combined into a *rule memory*, and Bayesian networks can be decomposed into idiomatic conditionals, such as those in Figures 32 and 33, that each represents an individual *network node* (in a Bayesian network); that is, a conditional or prior probability distribution over a child variable given its parents.

These examples also reveal that cognitive idioms can be based on restrictions of architecturally provided structures and/or combinations of multiple knowledge fragments; and that therefore the availability of a broad range of cognitive idioms depends on both the expressive generality provided by the cognitive language and the ability to integrate effectively across knowledge structures above the architecture (Rosenbloom, 2015). In general, different cognitive idioms arise from different combinations of properties of predicates and conditionals, and from the integration of different combinations of these at multiple levels of complexity.

As was mentioned in Section 5.1, the three memories in the cognitive architecture – perceptual, working and long-term – arise in Sigma as graphical idioms (over factor graphs). More specialized cognitive memories for such knowledge structures as rules, facts, episodes, constraints, images, perceptions, and actions all arise in Sigma as high-level cognitive idioms, each of which is composed from a combination of predicates and conditionals that themselves represent particular low-level cognitive idioms. In combination with Sigma's decision mechanism, cognitive idioms can also yield diverse forms of reasoning, whether a classifier (Figures 7, 32, 33, 44, 52 and 62), an HMM (Figures 14 and 18), a POMDP (Figures 15 and 17), problem-space search (Figures 25 and 26) or Theory of Mind (Figures 15 and 26). Likewise, in combination with Sigma's gradient-descent learning mechanism, cognitive idioms can yield a diversity of forms of learning, such as concept formation, clustering, perception modeling, action modeling, map learning (in SLAM), and reinforcement learning. Cognitive idioms, whether spanning memory, reasoning and/or learning are essential to enabling functional elegance above the cognitive architecture, much as graphical idioms are essential to enabling functional elegance in implementing the cognitive architecture.

Idioms that are well enough understood and sufficiently constrained may form the basis of *templates* that automatically create any additional required predicates and conditionals from the predicates that have been explicitly specified. A template enables Sigma to act much as if it has an architectural mechanism for the corresponding idiom – with the idiom being enabled automatically and no further knowledge needing to be added explicitly – but with processing still occurring via the knowledge above the architecture that defines the idiom. The template itself can conceivably be viewed as an additional architectural mechanism, possibly even as a special-purpose compile-time structure-learning mechanism, but it is typically a more minimal piece of special-purpose code than would be necessary to implement the idiom from scratch within the architecture. We are still working to fully understand the status and meaning of templates in Sigma, including whether they might eventually be replaced by an approach akin to *default rules* – that is, task-independent rules with which the system is initiated, such as those used in Soar for controlling weak-method search (Laird et al., 1987) – but this remains a topic for future work.

In the remainder of this section we will illustrate a variety of cognitive idioms related to memory, reasoning and learning – plus their associated templates when they exist – most of which would require special-purpose mechanisms or modules in a traditional cognitive architecture. The goal here is both to explain the basics of how such capabilities are defined and function within Sigma and to provide evidence concerning how far Sigma has been extended to date towards achieving grand unification in a functionally elegant manner.

For simplicity of exposition, the ideal would be to explain each group of idioms in its own subsection. However, memory and reasoning – at least reactive reasoning within a single cognitive cycle – cannot really be separated in Sigma, as it is not a von Neumann architecture in which the contents of a passive memory module are retrieved and then processed. Instead, each form of memory induces a form of reasoning that is an inextricable part of it, and which is indistinguishable from the notion of retrieval from the memory. Thus memory and reasoning will be described together in Section 6.1. The reasoning side of this will focus on reactive forms, which are those most closely tied to memory, but will also include bits of deliberative and reflective reasoning. For the rest, an understanding of how Soar does deliberative and reflective reasoning, plus the few examples from earlier sections of how Sigma goes beyond Soar in this area, will need to suffice here. Learning also has conflation issues with memory and reasoning in Sigma, but it is somewhat more decoupleable, and so will be described separately in Section 6.2.

6.1 Memory & Reasoning Idioms

Memory idioms are based on characteristic LTM structures – that is, on particular forms of conditionals – plus possibly particular forms of predicates and whether they include WMFNs, LTMFNs and PBFNs. A reasoning idiom then is simply the distinct form of processing that results from such a combination of structures and functions.

The most basic memory distinction in cognitive science, beyond those already captured in Sigma's cognitive architecture, is between procedural and declarative memory. This is different from, although not completely unrelated to, the distinction in artificial intelligence between procedural and declarative semantics, with the essence of the memory distinction being that the former drives behavior while the latter provides knowledge about the world. Rule memories provide the classical form of procedural memory in cognitive architectures (Anderson et al., 2004; Laird, 2012; Meyer and Kieras, 1997), but such memories can take on other forms as well, such as plans, cases and hierarchical skills (Fikes et al., 1972; Veloso and Carbonell, 1993; Langley and Choi, 2006) or even programs (Goodman et al., 2008; Goertzel et al., 2014). Declarative memories traditionally take on the form of semantic memories (facts and/or concepts) and/or episodic memories (past history) (Vere and Bickmore, 1990; Laird, 2012), but they too can take on other forms, including full logics (Wang, 2007; Domingos and Lowd, 2009; Goertzel et al., 2014).

The procedural-versus-declarative distinction is useful here as a top-level of organization across many of Sigma's idiomatic memories, but after exploring it in some detail in Sections 6.1.1 and 6.1.2 we will examine a combined idiom for trellises in Section 6.1.3, followed by two additional idioms that don't map as neatly onto this distinction – for imagery in Section 6.1.4 and distributed vectors in Section 6.1.5 – before concluding with some general discussion in Section 6.1.6.

6.1.1 Procedural Idioms

As mentioned in Section 4.4.4, one key aspect of the distinction between procedural and declarative memory in Sigma is that the former is characterized by conditions and actions, whereas the latter is characterized by condacts and a function. It is the unidirectionality of information flow in conditions and actions that provides the basis for behavior. This unidirectionality is also one of the key reasons why logical implications are not quite the same as the rules that are typically supported in cognitive architectures.

The full memory idiom for *standard rules* – which match conditions to working memory and make changes back to it according to how actions are instantiated by variables bound in the conditions – is characterized by three traits in Sigma: (1) conditionals are based (only) on conditions and actions; and predicates are (2) closed-world and (3) universal. It is thus an idiom based on particular forms of LTM structures plus WM functions. The form of idiomatic rule-based reasoning that emerges from these traits proceeds by matching and firing rules in parallel, and chaining across the rules that fire within a single elaboration phase.

Figure 31(a) exemplified a standard rule in Sigma, with its accompanying universal, closed-world `Above` predicate. Figure 64 shows a similar kind of rule that uses an affine transform to implement the (deterministic) operator `left` in a 1D grid, such as the one in Figure 10. However, this particular rule violates trait 3 by including an action for a unique predicate – `Location(state:state x:x!)` – which indicates that there is only one correct location resulting from moving left. The results of such an action are not available in WM until after a decision occurs, rather than enabling further chaining within a single elaboration phase. This conditional is therefore best considered as an instance of a separate idiom for *transition functions* or *action models*, which are still rules in a generic sense but not standard rules.

```
CONDITIONAL Move-Left
   Conditions: Selected(state:s operator:left)
               Location(state:s x:x)
   Actions: Location(state:s x:x-1)
```

Figure 64: Grid conditional for moving left.

Even though rules have dominated discussions of procedural memory in cognitive architectures, the essence of any procedural memory is simply a combination of *actions* with *control* over their execution. In ACT-R, rule actions provide the necessary actions, with match of buffers to rule conditions providing one aspect of control, and levels of activation providing the other. In Soar, rules also provide actions, which in this case non-monotonically alter working memory. Soar's parallel rule system enables conditions to provide the sole means of control in choosing which actions to execute. In Sigma, actions make persistent – that is, latched – non-monotonic changes to working memory. Conditions provide one form of control on action execution, but condacts and functions may also provide such control, thus potentially extending what should be considered procedural memory in this generic sense within Sigma.

```
CONDITIONAL Transition
   Conditions: State(state:s)
               Location(state:s x:x)
               Selected(state:s operator:o)
   Condacts: Location*Next(state:s x:nx)
             Location-Transition(x:x operator:o
                                 nx:nx)
```

Figure 65: Conditional for a 1D transition function.

One such extended procedural idiom is *probabilistic transition functions*, which include a function as part of the context for a transition function, and which thus violate trait 1 for standard rules. Figure 65 shows such a conditional for modeling movements in a 1D grid during SLAM. The conditional probability distribution over the next location given the current one is stored in the LTMFN of the `Location-Transition` predicate. Rather than an action for the closed-world `Location` predicate, there is now a condact for the open-world `Location*Next` predicate – yielding a second type of violation of trait 1 – because prediction mode is enabled and bidirectional processing will be required between the function and this predicate for learning. This conditional thus not only violates trait 3, but it also violates trait 1 in two ways. Via the condact, information about the next location is gathered in the (non-latched) working memory of the `Location*Next` predicate – that is, in its WMVN – before being gated for selection to the `Location` predicate, which then during the adaptation phase latches the new distribution in place of the previous one. So this has the same effect on the `Location` predicate as if an action had been used in the conditional for it, but with an intermediate representation of it also being created in the `Location*Next` predicate.

Figure 66 shows a similar transition function that has been used in an isolated-word speech-recognition HMM based on a reusable generic time slice (Figure 18). The function, which is stored in the LTMFN of the `Phone-Transition` predicate, represents the joint conditional probability of the next word and phone given the currently selected word and phone.[7] In Figure 18, this corresponds to the horizontal arrow. As in SLAM, this transition function leverages prediction mode, so there is a condact for Phone*Next rather than an action for Phone. What is different, though, is that no current operator is tested, as this transition function is part of perception, with no associated operators. Thus, although this is still a transition function, it isn't an action model.

```
CONDITIONAL Acoustic-Transition
   Conditions: State(state:s)
               Phone(state:s word:w_p phone:p_p)
   Condacts: Phone*Next(state:s word:w_c phone:p_c)
             Phone-Transition(word:w_p phone:p_p
                              nword:w_c nphone:p_c)
```

Figure 66: Conditional for a transition function from a word-recognition HMM based on a reusable generic time slice.

A template has been defined in Sigma for probabilistic transition functions that automatically turns on prediction mode if it isn't already on, to yield *Next predicates, and creates a transition conditional for each unique closed-world *state predicate* that is defined (where a state predicate is a predicate that includes an argument for the state). If the Selected predicate is defined, a condition for it is also included in the conditional, to convert the transition function into an action model.

In Soar, rules (and transition functions) reside within the reactive control level, whereas in ACT-R they are at the deliberative level, with one rule selected for execution each cognitive cycle. Soar also has a distinct notion of actions at the deliberative level – problem-space operators – with preferences retrieved from rule memory providing control for them (Section 5.3). Rules in Soar thus play an important role in procedural memory at both the reactive and deliberative levels, with standard rules and their actions at the reactive level and *control rules* guiding operator selection at the higher level. Sigma too augments its standard rules at the reactive level with a Soar-like higher level of operators and control rules. Control rules in Sigma are based on the same procedural idiom as transition functions: they are like standard rules in form, but their actions are based on unique rather than universal closed-world predicates. These actions may be based on the Selected predicate, for choosing the next operator to execute, or on any predicate that requires a choice.

This control-rule idiom leverages how distributions over unique closed-world predicates in Sigma can replace Soar's special-purpose preference language for operator choice. Figure 39, for example, showed a task-independent control rule that converts policies, in the form of Q functions, into distributions over the Selected predicate. The selection rule in the classification scenario – Figures 16(b) and 31(b) – is also such a control rule, although it strays even further from a standard rule by violating trait 2 to include an open-world condition that enables it to extract results from the classifier (a part of declarative memory that will be discussed shortly). In contrast, Figure 67 shows a task-dependent control rule for the Eight

```
CONDITIONAL Goal-Right-Best
   Conditions: State(state:s)
               Board(state:s x:x y:y tile:t)
               Board(state:s x:x+1 y:y tile:0)
               Board*Goal(state:s x:x+1 y:y tile:t)
   Actions: Selected(state:s operator:t)
   Function: 1
```

Figure 67: Eight Puzzle control rule for establishing that moving a tile to the right is *best* when it is its goal location.

[7] A *phone* is similar to a *phoneme*, but the former is a physical realization of a sound while the latter is an idealized sound.

Puzzle that assigns the highest possible rating to moving a tile into the blank cell to its right when that is the tile's desired location.

The most important result in general from this discussion of Sigma's procedural memory is that it is not limited to standard rules, and thus to the three traits listed above for them. It may, as we have seen, violate trait 3 by including actions for unique predicates, or violate trait 1 by including functions and/or condacts. It may also violate trait 2 by including a condition or action over an open-world predicate. These "violations" expand the space of procedural memory idioms – while still remaining focused on actions and their control – to span standard rules, control rules, (probabilistic) transition functions, and likely more.

6.1.2 DECLARATIVE IDIOMS

Declarative memory in Sigma was based in Rosenbloom (2010) on the antithesis of the three traits characterizing rules: (1) conditionals are based on (only) condacts and a function; and predicates are (2) open-world and (3) unique. In this case, traits 1 & 2 still seem essential: trait 1 to specify the relevant variables and relations plus the functional values over them (whether Boolean or fully numeric); and trait 2 to enable multiple pieces of evidence to be combined conjunctively (i.e., multiplicatively). However, trait 3 is perhaps best considered as one side of a further distinction within declarative memory, in analogy to the role trait 3 was eventually concluded to play in procedural memory. If we violate trait 3, and thus use universal predicates, we have an idiom for constraints, in which any number of the domain elements may be valid for the variables (Figure 34). Multiple such constraints yields a constraint network, or what could be called a *constraint memory* in the terminology used here. Bidirectional processing over the constraint conditionals ensures that all of the functions in the network/memory propagate appropriately.

Although a constraint memory of this type is not typically considered for inclusion in cognitive architectures – as opposed to the kind of soft constraint networks over which neural network models often seek to optimize (O'Reilly and Munakata, 2000; Sun, 2006; Eliasmith, 2013) – we are not aware of any direct evidence as to whether or not such a memory might play a role in human cognition. Still, in the generic cognition sense, it does make sense to consider such a constraint memory as providing a declarative memory idiom in Sigma.

With trait 3 intact – that is, with unique predicates – we are in the world of soft/probabilistic constraints, where there are distributions over the domains of the variables and the goal is to identify the best values for these variables. The reason this trait was originally considered essential to declarative memory was because it is central to the memory idioms for the two classical forms of declarative memory: semantic memory and episodic memory.

A sample of the conditionals involved in *semantic memory* – as mapped onto a naïve Bayes classifier (Figures 7, 44 and 52) – was seen in Figures 32 and 33, with the full factor graph for a portion of this classifier in Figure 62. An abstract view of the factor graph for the full classifier from the classification scenario is shown in Figure 68. Only the PBFN,[8] LTMFN and join/beta factors are shown. Evidence here appears in the perceptual nodes, with messages passing bidirectionally to generate a distribution over the concept in the forward direction – that is, towards the concept – and predicted distributions over the attributes in the backward direction.

[8] In earlier versions of Sigma, this was a WMFN, but since then the perceptual buffer was added and WMFN nodes for open-world predicates were eliminated.

This idiom for semantic memory induces classification reasoning as well as a form of synchronic prediction. In a bit more detail, it provides a complete partial match capability, where evidence can be provided for any of the attributes, as well as for the concept itself, with all those unspecified being predicted. For example, given perception that the `Color` is `silver`, plus a standard set of LTM functions for the prior on the `Concept` and the conditional distributions of the attributes given the `Concept`, the predicted values (and their probabilities) were `Concept=walker` (.90), `Weight=15.67` (expected value via `$` rather than best value), `Mobile=true` (.95), `Alive=false` (.95), and `Legs=4` (.99). The most significant examples of semantic memory so far implemented within Sigma have all been in the context of language processing, covering part of speech tagging and word sense disambiguation (Rosenbloom et al., 2013), as well as utterance classification (Ustun et al., 2015).

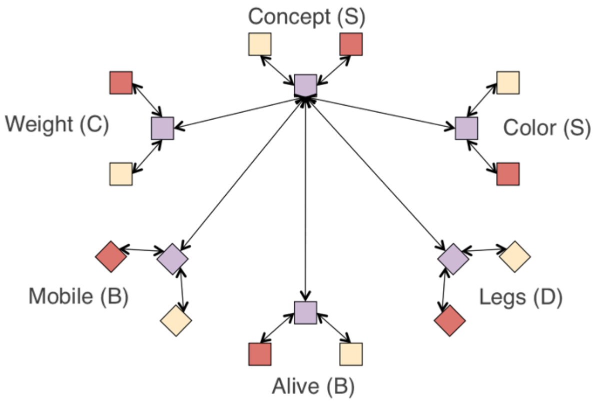


Figure 68: Abstraction of (naïve Bayes) semantic-memory factor graph, with (S)ymbolic, (D)iscrete, (B)oolean and (C)ontinuous types.

Digging more deeply into the naïve Bayes idiom for semantic memory reveals that it is itself decomposable into two smaller idioms, one for a prior distribution over the concept and the other for a conditional distribution over an attribute given the concept (Figures 32 or 33), both of which are usable independently. We can also see that although this overall naïve Bayes idiom is limited by the strong independence assumptions underlying such classifiers, Sigma as a whole is not limited in this manner. For example, the conditional in Figure 36 showed a (uniform) conditional distribution for the Q value (the child variable) given the combination of the location and the operator (the parent variables). Thus it is conceivable that Sigma's semantic memory could be more flexible in terms of its underlying independence assumptions – and might even learn which forms of independence do or do not exist – but this remains to be explored. There is also no inherent limit to a one-level hierarchy, although initial experiments with multiple levels – as in deep learning/networks (LeCun et al., 2015) – have not yet generated anything particularly of note.

Perceptual memory is typically considered as distinct from semantic memory, and clearly not a part of declarative memory. As a subcognitive skill, perception seems in many ways even more an antithesis of the symbolic facts and concepts in semantic memory than is (cognitive) procedural memory, with the notion of "declarative" usually even carrying along with it the connotation of a symbolic relational structure about which one can explicitly reason. Yet, in Sigma these distinctions largely disappear, even when focused, as in this section, on the long-term form of perceptual memory in LTMFNs rather than on the transient form of it in PBFNs. The perceptual memories used currently in SLAM, where the current location is to be determined indirectly from object perception, and in word recognition, where the current phone (the hidden state variable in the HMM) is to be determined from perception of spectral labels, are based on conditionals that are nearly isomorphic to the one in Figure 33, albeit with state arguments and a condition for the state added. Figure 69, for example, shows the conditional that relates objects to locations, with the resulting map – in the form of a conditional probability distribution over the object given the location – stored in the LTMFN for the Object-Location predicate. Figure 70 further shows the corresponding conditional for isolated-word recognition, which encodes the conditional probability of the spectral label given the word and phone, and which corresponds to the vertical arrow in the single-slice HMM in Figure 18. (Such conditionals can actually be thought of as inducing an instance of that diagram for each word). Thus, in Sigma's terms, such perceptual memories appear as fragments of semantic memory. The attributes and concepts here may be numeric rather than symbolic, but that turns out to be just as true a possibility for semantic memory in general.

```
CONDITIONAL Object-Location-Map
   Conditions: State(state:s)
   Condacts: Object(state:s value:o)
             Location*Next(state:s x:x)
             Object-Location(value:o x:x)
```

Figure 69: Map conditional for conditional distribution over objects given locations.

As with transition functions, a template has been defined for perceptual memories. It enables prediction mode if it isn't already enabled, and then creates for each perceptual predicate a conditional that has a condition for the state plus condacts for the perceptual predicate, all of the *Next predicates, and a new memorial predicate whose LTMFN stores the conditional probability of the unique variables in the perceptual predicate given the variables in the *Next predicates. Ultimately, a smarter approach will likely be required for determining upon which *Next predicates the perceptual predicates should be conditioned, but the approach described here has been sufficient to date.

```
CONDITIONAL Acoustic-Perception
   Conditions: State(state:s)
   Condacts: Observation(label:o)
             Phone*Next(state:s word:w phone:p)
             Observation-Phone(label:o word:w phone:p)
```

Figure 70: Perceptual conditional for word recognition.

An even simpler idiom for perceptual memory dispenses with conditionals entirely, in favor of a single perceptual predicate with an LTMFN. The concept in the classification scenario, as shown at the top and right of Figure 62, is of this sort, as is the perceptual memory for the visual search task in Section 4.3.1, where there is simply an Image-Color predicate with an LTMFN for the distribution over the color given the location in the 2D visual field. Similarly, in work on reinforcement learning (Section 6.2.5) there is a Reward predicate with an LTMFN for the distribution over the reward given the state (in this case, a location in a grid). In principle, this simpler idiom seems like it need not be limited to perceptual memory, and perhaps could provide a simple idiom for semantic memory or even declarative memory as a whole; however, if the LTMFN is to be learned based on gradient descent then it must either receive input from perception – that is, from a corresponding PBFN – or from the rest of the graph (via structure defined by an encompassing conditional), and so the predicate must either be perceptual or be connected via conditionals to other portions of the graph.

If we now move from semantic (and perceptual) memory to the other traditional form of declarative memory – *episodic memory* – the main differences between the two cognitive idioms at a high level are that episodic memory uses: (1) a concept corresponding to the episode number, with the original concept becoming just another attribute (Figure 71); and (2) *max* rather than integral for summarization so as to retrieve all of the attribute values for the single most appropriate past episode rather than the best individual attribute values across all episodes. Episodes here correspond to cognitive cycles, based on: an architectural Time predicate with a single argument whose value corresponds to the number of cognitive cycles that have elapsed, and attributes that are based on closed-world state predicates. As with all naïve Bayes classifiers, there is a prior distribution over the concept, which is Time here, and conditional distributions over all of the attributes given the concept.

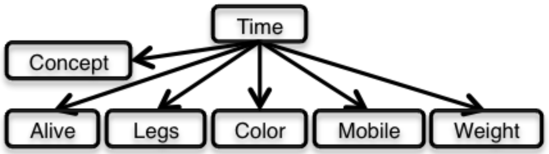


Figure 71: Naïve Bayes classifier for episodic memory.

This is roughly how episodic memory was originally implemented in Sigma (Rosenbloom, 2010), with a special-purpose episodic learning mechanism later added for it. However, more recent work has shown how, with a more elaborated memory structure and some parameter tuning, gradient descent could yield episodic learning in a manner that is uniform with the other forms of learning in Sigma (Rosenbloom, 2014). The more elaborated memory structure stems from the need to distinguish between the past and the present: episodic learning depends on what is true now; episodic selection – that is, deciding which episode to retrieve – depends on matching what is true now to what was true in the past; and episodic retrieval – that is, retrieving the attribute values associated with the episode – depends on what was true in the past. Semantic memory lacks this explicit historical/autobiographical aspect, being essentially timeless. In episodic memory, what is true now is represented in the WMFNs for closed-world selection state predicates. Each such predicate is assumed to define a single attribute with respect to episodic memory. These are therefore not quite the same predicates of the same name used in the classification scneario, as those are open-world. Instead these are selection predicates, like the use of Selected in the classification scenario, but here for the concept and each feature. As with the open-world *Next predicates for prediction (Section 5.4.1), open-world predicates with a suffix of *Episodic are added in episodic memory to represent what was true in the past. Three conditionals are then required per attribute/predicate, one each for learning, selection and retrieval. The key to making this work is *tying* the functions across these conditionals, as introduced in Section 4.4.3. The details can be found in Rosenbloom (2014).

	Concept	*Color*	*Alive*	*Mobile*	*Legs*	*Wgt.*
T1	walker	silver	false	true	4	10
T2	human	white	true	true	2	150
T3	human	brown	true	true	2	125
T4	dog	silver	true	true	4	50

Table 2: Sequence of four full instances.

For illustration, Tables 2 and 3 show a sequence of eleven episodes, starting with four that are fully specified and followed by seven that are partially specified (Rosenbloom, 2014). The latter seven are functionally "query" episodes, but they are themselves also learned as episodes, as there is no architectural distinction been training and testing trials.

	Queries	Best
T5	*Concept*=walker	T1
T6	*Color*=silver	T4
T7	*Alive*=false, *Legs*=4	T1
T8	*Alive*=false, *Legs*=2	T3
T9	*Concept*=dog, *Color*=brown	T4
T10	*Concept*=walker, *Color*=silver, *Alive*=true	T1
T11	*Alive*=false	T8

Table 3: Sequence of seven partially specified "queries".

There is one curve in Figure 72 for each of the query episodes, graphing the posterior distribution over the previous episodes[9] during the episode when the query appears (and thus when it acts as a cue for episodic retrieval). These distributions arise from a (multiplicative) combination between

[9] The curves actually start two episodes back, as it takes one cognitive cycle, and thus one episode, for learning to incorporate episodes into the appropriate LTMFNs.

the temporal prior learned across the episodes and the (partial) match between the present attribute values and those stored in the episodes. The episode retrieved for each of these queries can be seen in the final column in Table 3.

A template for episodic memory enables it to work automatically with no manual additions of knowledge. For each unique, closed-world state predicate that makes a selection, it generates an open-world `*Episodic` predicate plus the appropriate learning, selection and retrieval conditionals.

In summary on declarative memory, the primary focus has concerned idioms that are based on classifers, mostly of the naïve Bayes sort, to span semantic memory, perceptual memory and episodic memory. Other forms of classifiers are possible but have been explored less in Sigma. Declarative memories not based on classifiers are also conceivable, but so far unexplored.

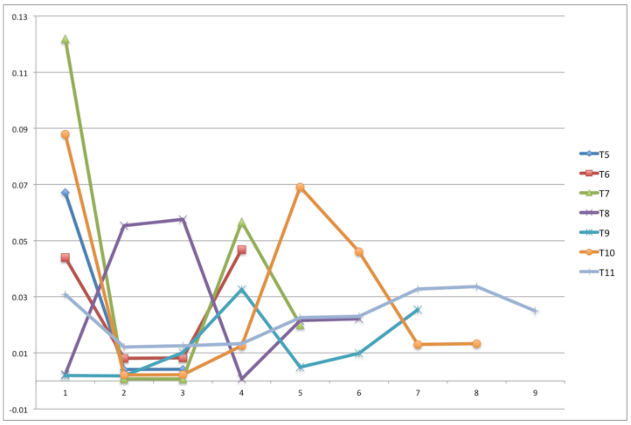


Figure 72: Posterior distributions for the seven partially specified “query” episodes.

6.1.3 MIXED IDIOMS: THE TRELLIS

One important way to move beyond the distinction between procedural and declarative memories (and idioms) is to consider memories (and idioms) that combine the two. For example, when the idioms for probabilistic transition functions and perceptual memories are combined, a new cognitive idiom arises that spans procedural and declarative memory. In particular, it yields a generalization over the generic single-slice HMM found in Figure 18 that includes the option for conditioning on the action performed (Figure 73). This combined idiom subsumes HMMs, but also covers a broader set of *trellis graphs*, enabling it to be used for both SLAM and word recognition. In SLAM, exploiting this pair of templates implies that only two conditionals, to drive a random walk around the environment, need be specified explicitly. To complete the processing in word recognition, only one conditional need be added, which combines a condition over the open-world `Phone*Next` predicate; an action for a unique closed-world `Word-Selected` predicate; and max summarization to select the best word from this predicate. (This conditional itself can be thought of either as representing a hybrid between procedural and declarative memory or as an additional procedural idiom.) The combined trellis idiom/template, as well as either of its subidioms, may also be useful elsewhere. For example, the probabilistic transition function idiom/template is also leveraged in the reinforcement learning work that will be described in Section 6.2.5.

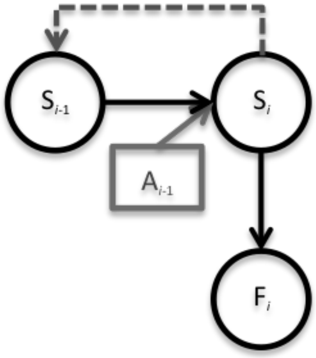


Figure 73: Generic single-slice trellis with an optional action.

These single-slice trellis graphs are inherently deliberative, spinning out in a forward direction across a sequence of cognitive cycles, although they can also support reflective variations as necessary, such as that shown in Figure 26 for the Ultimatum Game. Reactive within-cycle trellises are also possible in Sigma, and may be either unidirectional or bidirectional. The key to reactive trellises is for the probabilistic transition functions to be able to chain without requiring a selection, and thus a decision. For bidirectional graphs, the transition functions must themselves be bidirectional, and thus based on pairs of condacts. Figure 15 showed a POMDP for the Ultimatum Game as a unidirectional reactive trellis, whereas Figure 17 showed (unidirectional) interactions among three bidirectional reactive trellises, for perception (a CRF), localization (part of SLAM) and decisions (a POMDP). Like deliberative trellises, reactive trellises can be considered as hybrids between procedural and declarative memory, or possibly even as occupying a netherland between them.

Theory of Mind (ToM), such as in the Ultimatum Game, involves forming models of others' minds and generating expectations about their behavior based on these models, to enable effective decisions in social settings (Whiten, 1991). The reactive and reflective POMDPs for the Ultimatum Game in Figures 15 and 26 thus also reflect reactive and reflective idioms for ToM in Sigma. Other forms of ToM reasoning have also been demonstrated in Sigma – such as the computation of the Nash equilibrium for the Prisoner's Dilemma – that involve different ToM idioms (Pynadath et al., 2013), but these will not be further detailed here.

Trellises in general, but most particularly unidirectional ones at the reactive and higher control levels, show the leverage that is possible when "different memories" – here procedural and declarative – are combined not by loosely coupling monolithic modules but by tightly coupling small fragments of each into a new hybrid fragment, yielding both a new hybrid idiom and a new form of idiomatic memory. Such trellises are not possible in conventional cognitive architectures such as Soar and ACT-R without the explicit addition of new architectural modules. Typically these additional modules must also be implemented in a manner that is inhomogeneous with the core architecture because they require controlled forms of subsymbolic processing.

6.1.4 IMAGERY IDIOMS

One major memory (and reasoning) idiom that has not yet been covered yields *imagery memory* (Rosenbloom, 2011b, 2012b). It is one of the simplest idioms in Sigma because it requires no specific conditionals. Essentially, a form of mental imagery is implicated anytime there is a predicate with at least one universal numeric argument, either discrete or continuous, and one unique argument. This could be 1D, 2D, 3D or more, depending on the number of universal arguments. The `Location` predicate implicit in Figure 10 and explicit in Figures 36, 39, 64, 65 and 69 is a discrete 1D fragment of imagery memory. The `Image-Color` predicate implicit in Figures 28 and 30 is a discrete 2D fragment of imagery memory. The `Board` predicate for the Eight Puzzle, as introduced in Section 4.4.3, is a continuous 2D fragment, as is the `Image` predicate used in Figures 37, 40 and 54. Perhaps more surprisingly, the `Q` predicate in Figures 36 and 39 that represents a policy for operator selection yields a discrete 1D fragment of imagery memory, as it has a universal discrete numeric argument like the one used in the `Location` predicate, plus a unique argument for the utility.

Even perceptual memories are effectively fragments of imagery memory, as long as they have the right combination of argument types. For example, both the perceptual field in visual search and the map function in SLAM are part of mental imagery, whereas the acoustic function currently used in word recognition is not, because the first two are based on discrete numeric locations, while the latter is based on symbols representing acoustic labels. Similarly, the perceptual `Reward` predicate mentioned earlier in this section is part of imagery memory when the reward is conditioned on a numeric state and is not when it is conditioned on a symbolic state (unless the definition of what it means to be an image is further extended). A hybrid state might yield a memory fragment whose category is ambiguous, but whose meaning should still be straightforward.

As with all of the other idioms described in this section, reasoning over imagery occurs via conditionals: in particular, ones based on affine transforms for translation, scaling and reflection (Figure 40); filters to extract and weight parts of images (Figure 39); function products to conjunctively combine images, such as in computing the intersection or overlap among them (Figure 74); function summarization to disjunctively combine images, such as in computing their union (Figure 75); and negation/inversion to assist in computing the differences between images (Figure 76). Of these architectural capabilities implicated in imagery reasoning, only one was explicitly developed in support of imagery reasoning: affine transforms. Surprisingly, this classical form of imagery reasoning also turned out to support a primitive form of architectural arithmetic over numeric variables, where translation performs addition and scaling/reflection performs multiplication.

```
CONDITIONAL Overlap-0-3
   Conditions: Image(o:0 x:x y:y)
               Image(o:3 x:x y:y)
 Actions: Overlap(i:1 x:x y:y)
```

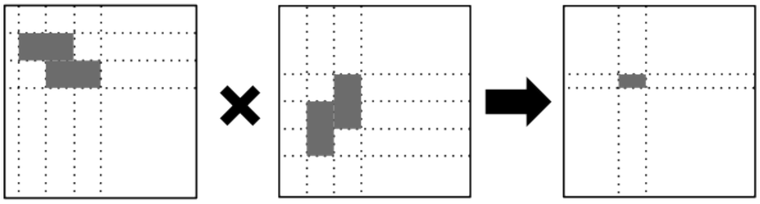
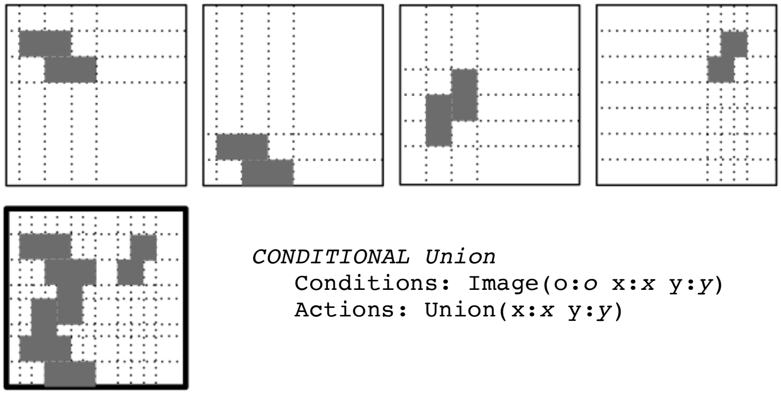


Figure 74: Computing the overlap between images 0 and 3.



```
CONDITIONAL Union
   Conditions: Image(o:o x:x y:y)
   Actions: Union(x:x y:y)
```

Figure 75: Combining four objects into a single new object.

6.1.5 DISTRIBUTED-VECTOR IDIOMS

The last idiom that will be covered in this subsection is for *distributed vector memory* (Figure 13) and its associated reasoning (Ustun et al., 2014). This idiom is like the one for imagery memory in only constraining the predicates. However, in this case they are limited to *vector predicates*. Sigma was able to support distributed vectors to some extent prior to the development of the vector predicate type, via unnormalized unique predicates with a single discrete argument, but small tweaks to the ways normalization, action

```
CONDITIONAL Left-Edge
   Conditions: Union(x:x y:y)
               -Union(x:x-.0001 y:y)
   Actions: Left-Edge(x:x y:y)
```

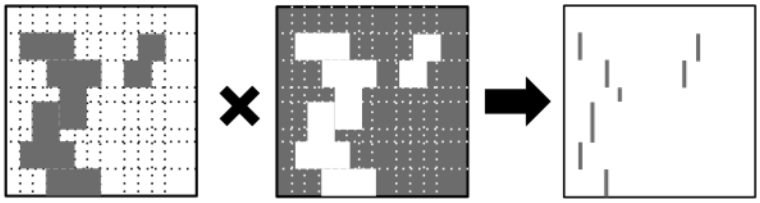
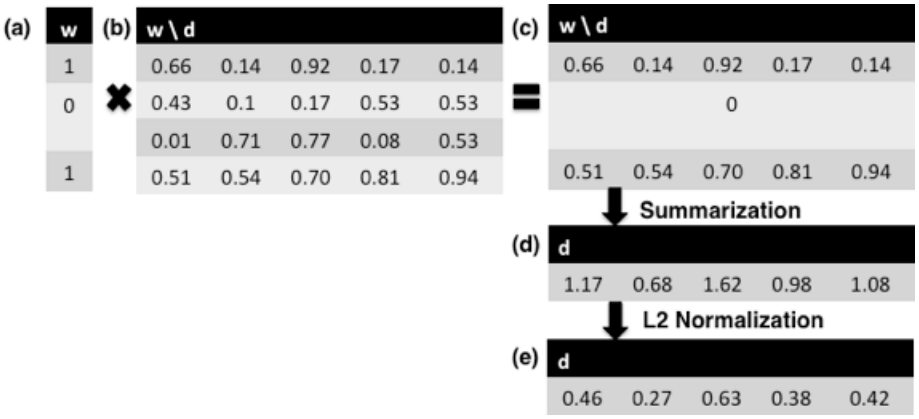


Figure 76: Computation of the left edge of a complex object via translation, negation and multiplication.

combination, and learning would work for vectors improved Sigma for this form of memory (Section 4). With these changes, vector summation occurs via action combination, vector binding continues to occur via multiplication, and vector comparison occurs via cosine similarity (based on a combination of multiplication, summarization and vector normalization).

Figure 77, for example, illustrates how knowledge about which other words have co-occurred with a particular word in a fragment of text is converted into a single new vector representing this context (Ustun et al., 2014). It starts with (a) a local Boolean vector over the vocabulary that has a 1 for each co-occurring word and a 0 for all other words, and (b) the lexicon, represented as a matrix that maps local word representations into distributed representations. The product of these two (c) yields non-zero rows in a matrix for only the co-occurring words, a form of multiplicative selection. Summarizing out the rows via integral yields a single summed vector (d), which is then vector normalized (L2) to yield the context vector (e).



w
1
0
1

w \ d				
0.66	0.14	0.92	0.17	0.14
0.43	0.1	0.17	0.53	0.53
0.01	0.71	0.77	0.08	0.53
0.51	0.54	0.70	0.81	0.94

w \ d				
0.66	0.14	0.92	0.17	0.14
		0		
0.51	0.54	0.70	0.81	0.94

d				
1.17	0.68	1.62	0.98	1.08

d				
0.46	0.27	0.63	0.38	0.42

Figure 77: A small example, over a four-word vocabulary, of computing a context vector from a co-occurrence vector and the lexicon.

6.1.6 ADDITIONAL DISCUSSION

There are other memory and reasoning idioms in Sigma, such as heuristic search (Figure 25), but the way this currently occurs in Sigma is similar to the way it has always been done via decisions, impasses, reflection and metalevels in Soar, and thus not worth a further discussion here. Instead we'll wrap up this subsection with noticing how the memory and reasoning idioms introduced in this section together manifest functionally elegant grand unification above the cognitive architecture. From simple variations over the predicates and conditionals used, plus combinations of them, we have seen how everything from core perceptual activities to purely symbolic rule behavior is evidenced, along with a variety of more intermediate forms. Some architectural extensions have been implicated in the process, such as for affine transforms and vector predicates, but these are localized modifications that yield a missing fragment of generalized capability rather than new modules for their respective memories.

6.2 Learning Idioms

Sigma's learning idioms yield a diversity of supraarchitectural learning capabilities that are all (1) grounded in changes to LTMFNs, whether within predicates or conditionals, and (2) driven by gradient descent based on the "reverse" messages arriving at the corresponding factor nodes. Learning in Sigma is thus primarily a side effect of memory and reasoning – as was also the case earlier with chunking in Soar (Laird et al., 1986) – implying that learning idioms are typically side effects of memory and reasoning idioms. This section is therefore organized around the idioms introduced in Section 6.1, but with the discussion extended here to incorporate their consequences for learning. Memory idioms that do not involve LTMFNs will be passed over, as gradient descent can have no impact on them.

6.2.1 PROCEDURAL IDIOMS

With procedural idioms, we must immediately skip over both standard rules and basic transition functions because neither contains functions. Probabilistic transition functions, however, do contain functions and, in fact, require learning if the template is used because the LTMFN created for the transition function is initialized to a uniform distribution and thus can only have appropriate content through learning. *Action modeling* – that is, learning transition functions that provide models of actions – has been demonstrated via a transition conditional like the one in Figure 65 for a random walk in which there is perception of the location, but with noise in both this perception and in the outcome of the actions. For example, Table 4 shows the results from one experiment spanning 10K cognitive cycles in a 1D grid of size 8, with actions for `left`, `right`, and `stay`. Any attempt to move beyond the bounds of the grid here yields the same result as executing `stay`. The probability of perceiving the correct location is .9, the total probability mass allocated to the location perceived is .9 (with the rest allocated to adjacent locations) and there is a .9 probability of the selected action being the one executed. While the results in the table aren't perfect, a uniform distribution would yield just .125 while the minimum value for the correct location in the table is .58 and the maximum is 1.

	0	1	2	3	4	5	6	7
`left`	.97	.83	.86	.58	.60	.71	.69	.89
`stay`	1.00	.89	.74	.98	.87	.96	.89	.98
`right`	.80	.96	.75	.75	.71	.66	.61	.89

Table 4: Probability associated with the correct next location, given the action at the left of the table and the current location at the top of the table, as assigned by the learned action model from a random walk.

Action modeling without noise has also been investigated in the context of reinforcement learning (RL) (Rosenbloom, 2012a), where the probabilities of correct predictions quickly approach 1, and noisy, action-free *transition function learning* – as structured by the conditional in Figure 66 – has been demonstrated successfully for word recognition (Joshi et al., 2014). Action modeling has not been demonstrated successfully in SLAM, as without access to either an action model or location information it has not proven possible to bootstrap meaningful learning.

Learning for the simplest control rules, such as the Eight Puzzle control rule in Figure 35, which has a constant function and thus no variables, is not possible via gradient descent in Sigma because there are no child variables over which a distribution can be learned. However, *control learning* does become possible when a distribution exists over a variable for which a selection is to be made. For example, learning the full function underlying the contents of Table 4, which defines a distribution over the next location given the current location and action, amounts to learning to control the selection of the predicted next location, assuming that a selection will actually be made.

A more conventional example of control learning involves learning to select actions that yield better outcomes. *Reinforcement learning*, for example, learns a policy (or Q function): that is, a distribution over a utility variable given the state and action (Sutton and Barto, 1988). When combined with the generic conditional in Figure 39 that converts such policies into distributions over operators, a learned policy can help determine which actions are selected (Rosenbloom, 2012a). In Sigma, reinforcement learning is a complex, compound, cognitive idiom, and is the only learning idiom not directly tied to a single memory/reasoning idiom described in Section 6.1. As such, its further explanation will be deferred until Section 6.2.5, after all of the more primitive learning idioms out of which it is composed have been introduced.

6.2.2 Declarative Idioms

The first form of declarative learning idiom to consider is *constraint learning*. As introduced in Section 6.1.2, constraint reasoning in Sigma involves only universal variables, and thus there are no child variables on which gradient descent can operate. However, as was mentioned in Section 4.2.4, useful learning has proven possible here even with this basic assumption violation. But in the longer term, a valid learning algorithm is required for constraint memory and for all similar undirected functions in Sigma.

To understand *semantic learning* in Sigma, it helps to start by considering the abstract graph for semantic memory in Figure 68. For an attribute, messages from its PBFN and from the concept combine, via product and summarization at the corresponding beta/join node, to provide a message to this LTMFN that is usable as the gradient in learning the conditional probability of the attribute given the concept. For the concept, the message from its PBFN is combined with the messages from all of the attributes to provide the gradient at the LTMFN for learning the prior distribution, as shown by the top message in Figure 52. Work on learning such classifiers for two key aspects of natural language processing – word sense disambiguation (WSD) and part of speech (POS) tagging – has demonstrated *supervised concept learning* as well as aspects of *language learning* (Rosenbloom et al., 2013). For WSD, such learning yielded 78% correct on the Senseval-3 dataset and 70.3% correct on the Semcor dataset. For POS, such learning yielded 90.3% correct. Although this latter result was no better than the baseline of choosing the most common tag, it did at least show that this approach could learn to match this baseline.

This form of concept acquisition has also been extended in a straightforward manner to *unsupervised learning* (or *clustering*) (Rosenbloom et al., 2013). The key was to move the concept from its privileged position in the classifier to a position as just another attribute, as was similarly done for episodic memory in Figure 71, and then to add in the privileged position a variable that has one domain element per cluster to be found and neither evidence nor a prior. The LTMFNs here are initialized with random rather than uniform values to help break initial ties concerning how the different clusters respond to the initial attribute values that are experienced. Gradient descent over such a structure yields unsupervised learning in a manner that resembles standard clustering techniques such as *expectation maximization* (Rosenbloom et al., 2013). In particular, evidence for attribute values engenders distributions over the available clusters, which bounce back to generate expectations for the values of the (unspecified) attributes. Learning occurs for the conditional probabilities in the LTMFNs in service of aligning expected and actual values. Such an approach can learn a correct model for exclusive or (XOR), whereas a one-level supervised naïve Bayes classifier cannot.

A partial template exists for semantic memory that can automatically generate a supervised or unsupervised classifier based on a specification of the concept and attributes, but the development of a full template that would automatically create an appropriate semantic memory given the predicates otherwise defined for task-oriented behavior remains for future work.

A template does however exist for perceptual memory that yields *perceptual learning*. For SLAM and word recognition this template has been used successfully to learn a map and an acoustic model, respectively. There are also a number of successful examples of more primitive perceptual learning, where there is a perceptual predicate with an LTMFN but no conditional or template: for example, for the `Image-Color` and `Reward` predicates mentioned in Section

6.1.2. Figure 78, for example, displays the expected values learned over the `Reward` LTMFN as a function of the location in a 1D grid, where there is an externally provided reward of `9` in cell `4` and `0` everywhere else (Rosenbloom, 2012a).

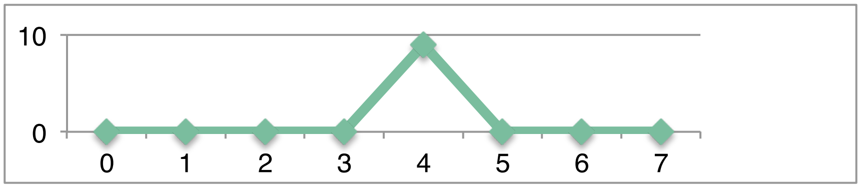


Figure 78: Expected values by location for the learned `Reward` LTMFN.

Episodic learning occurs over the elaborated classifier defined by the temporal prior plus the three conditionals per state attribute/predicate discussed in Section 6.1.2 (Rosenbloom, 2014). However, two additional tweaks have also been made: raising the learning rate for the episodic LTMFNs so as to increase the contrast for one-shot learning (given that each episode is seen only once), and switching from the subtractive form of normalization that is used elsewhere in gradient-descent learning to a divisive form. Divisive normalization, where individual values are divided by the sum/integral over all of the values, is a common form of normalization and was originally used in Sigma's learning algorithm, but it was eventually replaced by a subtractive version to help learning track the constraint surface more closely (Rosenbloom et al., 2013). The original divisive form is used in episodic learning so as to yield a temporal prior that decreases exponentially backwards in time. With this adjustment, and the fact that message bidirectionality implies that temporal learning occurs both when an episode is learned and when it is accessed, it was possible to map this temporal prior (Figure 79) onto ACT-R's notion of *base-level activation* without building in a separate activation model or base-level equation (Rosenbloom, 2014). In base-level activation, the activation of items in memory is based on a combination of their (exponentially decaying) recency and frequency. The inhomogeneity resulting from these two tweaks is troublesome conceptually, but resolving this is left for future work.

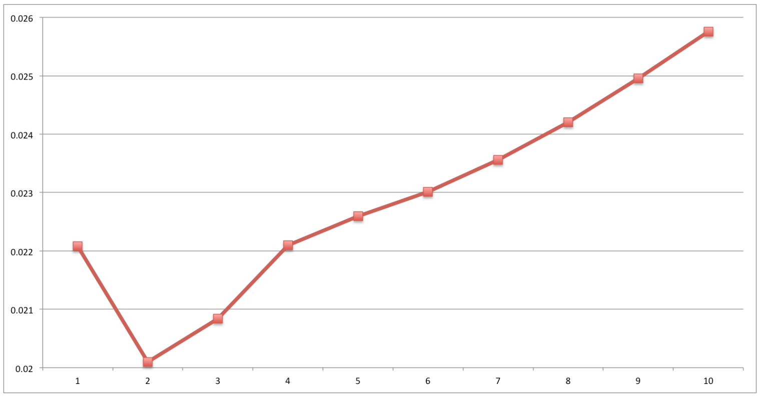


Figure 79: The learned temporal prior over the episodes so far experienced. It has a natural exponential shape from the learning of the episodes with divisive normalization, but portions of it also are bumped up as episodes are retrieved.

One standard optimization in purpose-built episodic memory modules within other architectures comes for free with episodic learning over piecewise-linear memories: as new episodes are acquired, new regions are required in an LTMFN only when the value of the attribute for the current episode differs from that in the previous episode. Otherwise, the boundary of the previous region is just effectively extended. Unfortunately, this optimization by itself is insufficient to make episodic retrieval sufficiently efficient when learning a large number of episodes, as the cost of episodic retrieval still grows linearly with the size of the LTMFNs due to the entire function for each attribute being passed as a message each cycle. Future work includes exploring how this cost may be reduced either via attentional focusing or via a form of *incremental message passing*, where only the changes to functions need be sent each cycle.

6.2.3 MIXED IDIOMS: THE TRELLIS

Full *trellis learning* combines the idioms and templates for transition functions and perceptual memories. The prime example of this to date is the simultaneous learning of transition and acoustic (perception) functions in a deliberative HMM for isolated-word recognition (Joshi et al., 2014), where an accuracy of 99.2% was achieved for a ten-digit vocabulary (compared to a baseline for random selection of 10%).

6.2.4 IMAGERY IDIOMS

Learning has not so far been explored in the work explicitly focused on representing and transforming mental imagery, but examples of *imagery learning* can be found in more naturalistic settings, where imagery simply exists in the context of work on other capabilities. One significant class concerns perceptual learning over numeric fields, as already described for maps in SLAM and the `Image-Color` and `Reward` predicates. A key non-perceptual example is shown in Figure 80, where a policy for the `Q` predicate is acquired via reinforcement based on the learned reward function in Figure 78 (Rosenbloom, 2012a). As is clear in this chart, the `Left` operator is preferred for locations to the right of the reward and the `Right` operator for locations to the left of it.

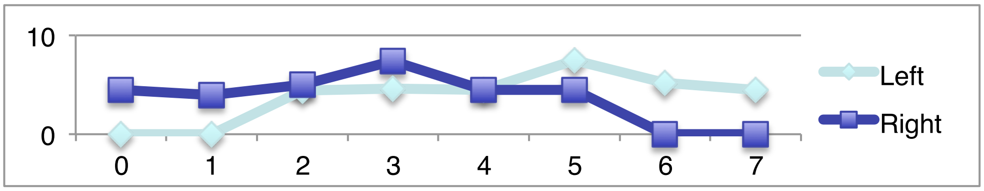


Figure 80: Expected values by location for the policy learned for the `Q` predicate given the reward function learned in Figure 78.

6.2.5 REINFORCEMENT-LEARNING IDIOM

Template-driven reinforcement learning in Sigma starts with perceptual learning of a reward function, as in Figure 78. This is then used as the basis for conjointly learning the projected future utility of states in the LTMFN for the `Projected` predicate, and the policy in the LTMFN of the `Q` predicate (Figure 80). Figure 81 shows the conditional that drives `Projected` learning, based on backing up the sum of the reward and the discounted projected value for the next state (from `Projected*Next`). This conditional uses an affine transform to discount the projected distribution (by *scaling*) and to add (by *translation*) the local reward to it in computing the `Projected` distribution from which it will learn. It furthermore weights this by the filtered `Q`

```
CONDITIONAL Backup-Projected
   Conditions: State(s:s)
               Location(s:s x:x)
               Location*Next(s:s x:nx)
               Selected(s:s operator:o)
               Projected*Next(x:nx value:np)
               Reward(x:nx value:r)
               Q(operator:o location:x value:[.05*q])
   Actions: Projected(x:x value:r+.95*np)
```

Figure 81: Conditional for backing up the `Projected` predicate in RL.

distribution, which converts the explicit distribution over utilities into an implicit distribution of functional values, as in Figure 39, so that if Boltzmann selection causes a poorly valued action to be tried, the projected future utility of the current state is not affected too greatly by any bad result that might occur. The conditional for backing up the policy is much like this one, but has an action for the `Q` predicate instead. Figure 82 shows the expected values learned for the `Projected` predicate for the same situation in which the `Reward` and `Q` predicates were learned in Figures 78 and 80. The distribution dips to `0` at the goal location because trials are terminated when the goal location is reached, and so there is no backup at that point from which to learn.

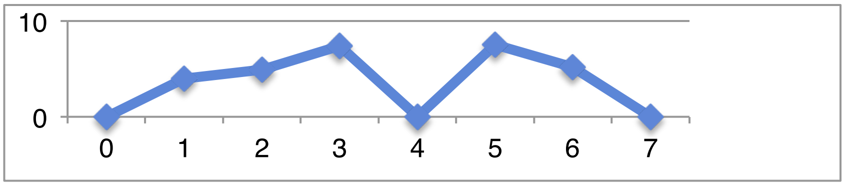


Figure 82: Expected values by location for the `Projected` future utility.

In RL, rewards are propagated backwards in time to learn policies. Reversing this – that is, propagating policies forward in time to learn rewards – yields *inverse reinforcement learning* (*IRL*) (Ng and Russell, 2000). IRL has been used in Sigma to learn, as reward functions, models of other agents – a key part of Theory of Mind – in a two-person negotiation game (Pynadath et al., 2014). The first step is to leverage simple perceptual learning to acquire a distribution over the other agent's operator choices given the current state, under the assumption – which works fine in the two-person games investigated so far – that agent choices are clear from their behavior. Then these action likelihoods are re-interpreted in terms of the expected values of Q distributions, and a softmax distribution is used to translate these Q values into a Boltzmann distribution. This distribution is then propagated forward to learn the projected reward in the resulting state. Because of the potential noise and error in this projection, each update is weighted by the likelihood of the observed operator, as in reinforcement learning. Finally, the constraint is exploited that in these domains rewards are only provided in terminal states, so that there is no projected utility at states where rewards are received. Therefore, the projected utility propagated forward to such a state can be assumed to be the reward for the state.

6.2.6 DISTRIBUTED-VECTOR IDIOM

The idiom for *distributed-vector learning* involves summing together, via action combination, the kind of context vector for a word computed in Figure 77 for co-occurrence information with an ordering vector for the word that is based on a variant of skip-grams, an alternative to n-grams in which nearby words are bound (by pointwise product) to unique vectors for their positions relative to the word of interest. This sum then becomes the gradient for learning a distributed representation of the word via the vector variant of gradient-descent learning described in Equation 5 (Section 4.2.4). Because of the large number of large distributed vectors used here, this idiom presently runs rather slowly in Sigma. We are investigating ways to speed this up in Sigma, but in the meantime a simulator, DVRS', has been used to generate results not currently feasible within Sigma (Ustun et al., 2014). Table 5 illustrates the results of this simulator for the task of finding similar words. In general, when comparing the results of this simulator versus the state-of-the-art Word2Vec system (Mikolov et al., 2013), we see cases where this approach does better, similar and worse (Ustun et al., 2014). There is clearly a core similarity between what the two approaches learn, but there is also enough difference that combining the vectors learned via the two approaches yields better results than either approach individually (Garten et al., 2015).

6.2.7 Additional Discussion

One key global result derivable from the set of learning idioms described in this section is that all memory and reasoning idioms, except those lacking directed functions – that is, those that have no function with a child variable – directly engender effective learning idioms, so that the scope of learning behaviors naturally grows with the scope of memory and reasoning behaviors. A second key result is that learning behaviors can range from the very simple, where there is just one function in isolation, up to complex forms such as trellis, episodic, reinforcement, and inverse reinforcement learning, where multiple functions in a complex relationship with each other must be learned in concert.

	language			***film***	
Context	**Order**	**Composite**	**Context**	**Order**	**Composite**
spoken	cycle	languages	director	movie	movie
languages	society	vocabulary	directed	german	documentary
speakers	islands	dialect	starring	standard	studio
linguistic	industry	dialects	films	game	films
speak	era	syntax	movie	french	movies
	business			***run***	
Context	**Order**	**Composite**	**Context**	**Order**	**Composite**
businesses	data	commercial	home	play	runs
profits	computer	public	runs	hit	running
commercial	glass	financial	running	pass	hit
company	color	private	hit	die	break
including	space	social	time	break	play

Table 5: Five nearest neighbors of four words based on context, ordering and the composition of the two.

6.3 Summary

Figure 83 summarizes the key idioms presented in this section, divided according to whether they concern memory and reasoning or learning. It should be clear that these idioms span many of the capabilities typically desired in a cognitive architecture, but in Sigma they have all appeared instead as cognitive idioms. There are important interactions among many of these idioms as well, whether it is combining multiple memory-and-reasoning idioms into a more complex reasoning structure, such as with trellises, or combining memory-and-reasoning idioms with corresponding learning idioms. Building towards virtual humans via complex combinations of such idioms is the core subject of the next section.

Memory and Reasoning Idioms

- Procedural Memory
  - Standard Rules
  - Control Rules
  - Transition Functions
    - Probabilistic Transition Functions
    - Action Models
- Declarative Memory
  - Constraint Memory
  - Semantic Memory
  - Perceptual Memory
    - Map Memory
  - Episodic Memory
- Trellis Memory (HMMs, POMDPs, etc.)
  - Reactive Trellises
  - Deliberative Trellises
  - Reflective Trellises
- Imagery Memory
- Distributed Vector Memory
- Problem Space (Heuristic) Search

Learning Idioms

- Transition Function Learning
  - Action Modeling
- Control Learning
  - Reinforcement Learning
- Constraint Learning
- Semantic Learning
  - Supervised Concept Learning
    - Language Learning
  - Unsupervised Learning
- Perceptual Learning
  - Map Learning
- Episodic Learning
  - Temporal Learning
- Trellis Learning
- Imagery Learning
- Inverse Reinforcement Learning
- Distributed Vector Learning

Figure 83: Primary cognitive idioms introduced in this section.

7. Combining Cognitive Idioms into Virtual Humans

In the previous sections, several combinations of cognitive idioms were seen, such as the combination of a probabilistic transition function with perceptual memory (and learning) to yield a generic single-slice trellis (Figure 73), and the combination of perception and localization with decision making (Figure 17). The combination of a POMDP with reinforcement learning and inverse reinforcement learning for agent modeling in Theory of Mind has also been demonstrated (Pynadath et al., 2014). However, a key goal for Sigma must always be to go beyond just combinations of small numbers of cognitive idioms, to full integration across all of the cognitive idioms that have so far been developed. This is a constantly receding goal, at least as long as new idioms are being developed, but one that can continually be approached by composing ever more ambitious combinations in the context of building *virtual humans* (*VHs*). Virtual humans are synthetic characters that can take the part of humans in a variety of contexts. Like intelligent agents and robots, they form one of the great integration challenges for research in AI.

Swartout (2010) has argued that the main goal for virtual humans is to look and behave as real people to the extent possible. Behaving like real people requires (1) using perceptual capabilities to observe the environment and other virtual/real humans in it; (2) acting autonomously in the environment based on what is known and perceived; (3) interacting in a natural way with both real and other virtual humans using verbal and non-verbal communication; (4) possessing a Theory of Mind to model both one's own mind and the minds of others; (5) understanding and exhibiting appropriate emotions and associated behaviors; and (6) adapting one's behavior through experience. Most critically, although most current virtual humans succeed in their aims with only a small fraction of these capabilities, ultimately this full range of capabilities must be integrated and must function coherently.

Broadly capable real-time virtual humans provide ideal testbeds for demonstrating and evaluating progress on Sigma's four desiderata, with "broadly capable" challenging the extent of its *grand unification*; "real-time" challenging its *sufficient efficiency*; and "virtual humans," with their need to exhibit human-like behavior in artificial systems, challenging its *generic cognition*. The fourth desideratum, *functional elegance*, implies a relatively unique path towards the construction of such virtual humans, where instead of a disparate assembly of modules (Hartholt et al., 2013), all of the required capabilities are constructed from a small combination of architectural mechanisms plus a hierarchy of cognitive idioms.

Two quite different, but both still fairly simple, virtual humans have been implemented in Sigma: a task-oriented *chatbot*, and a *virtual humanoid robot*. These two VHs occupy points in the virtual-human spectrum that would normally require quite distinct architectural support. The fact that both can be constructed in reasonable ways within Sigma demonstrates key aspects of all four of Sigma's desiderata.

7.1 Chatbot

The first attempt to build a virtual human in Sigma had modest goals, focused on replicating just the mind of a deployed, but cognitively rather simple, virtual human that was originally developed for the Immersive Naval Officer Training System (INOTS), a training vehicle for leadership and basic counseling by junior leaders in the U.S. Navy (Campbell et al., 2011; Ustun et al., 2015). The control structure of this existing virtual human is based on a branching – directed acyclic – network of states and utterances, with a statistical utterance classifier used to determine which choice, and thus also what response, to make at each point. In the original system, two separate tools were combined to make this work, with one providing the control structure and the other the utterance classifier (Campbell et al. 2011). In Sigma these two functionalities were mapped, respectively, onto: deliberative movement in a discourse problem space composed of operators for speaking and listening, and a reactive bag-of-words naïve Bayes utterance classifier. The first component involves the kind of deliberative Soar-like idiom on which this article hasn't focused, but it is built from a pair of simpler idioms that have been described, for action models and control rules. The second component comprises a standard application of Sigma's semantic memory idiom. This application can be viewed as an adaptation and extension of the classification scenario first introduced in Section 4, where the features are now words, the selection is of a sentence, and there is a deliberative sequence of such classifications + selections structured as a dialogue.

The goals for implementing this particular VH mind in Sigma were to show that simple minds of this sort could be created simply, and in an integrated fashion, and that they could then be extended as more complex intelligent behavior is required. These goals are related to Alan Kay's mantra "Simple things should be simple and complex things should be possible," and are intended to support typical customer requirements that start simple but then grow as it becomes understood what the initial system can and cannot do. It was in fact possible to demonstrate that a simple VH mind could be created simply and in an integrated fashion, in this case via a straightforward combination of a deliberative procedural idiom and a reactive declarative idiom, to yield a mixed supraarchitectural capability combination that spans multiple levels of the control hierarchy.

In a bit more detail, the VH control structure alternates whose turn it is to talk, with a `listen` operator selected when it is the human's turn and a `speak` operator selected when it is the VH's turn. When it is the human's turn, their typed utterance is classified as one from among a small set of standard utterances – typically between one and three – that have been predefined for the corresponding dialogue state. The selected standard utterance then feeds the transition function, to determine the subsequent dialogue state. Only one utterance is specified for each dialogue state in which it is the VH's turn to speak, so it is merely produced, and the next dialogue state then is a direct function of the previous one. This VH mind works in real-time, and can hold a dialogue like the one in the original system; however, due to extrinsic constraints its development was terminated before extensions to more complex behavior could be investigated.

7.2 Virtual Humanoid Robot

The second attempt to build a virtual human in Sigma started at a point that was already significantly more ambitious than the first attempt, and with the long-term aim of driving ever more aggressive combinations of capabilities. This initially yielded what can be considered an early form of an adaptive, interactive virtual human (Ustun and Rosenbloom, 2015), through a combination of (1) cognitive idioms for control rules, imagery, Theory of Mind, SLAM and (multiagent) reinforcement learning; and (2) a virtual physical embodiment via the SmartBody character animation system (Shapiro, 2011). The VH is adaptive not only in terms of dynamically deciding what to do via a rule-based decision framework, but also in terms of embodying both map and reinforcement learning. It is interactive both in terms of its (virtual) physical environment – through high-level perception and action – and its relationship to other participants, although the latter is still quite limited. Neither speech nor language was deployed in this virtual human, so social interaction was limited to constructing – actually learning with the help of RL – models of the self and others.

Embodiment occurred within a simulated convenience store for a *shoplifting* scenario, a civilian variant of the important class of *physical security systems* scenarios. In a typical shoplifting plot, offenders first pick up merchandise in the store and then try to leave without getting caught by any of the store's security measures. A simple grab-and-run scenario has been the initial focus, where the intruder needs to locate the desired item in the store, grab it, and then leave the store. The role of security is to detain the intruder before s/he leaves the store. A basic assumption is that it is not possible to tell what the intruder will do until s/he picks up an item and starts running. The security can immediately detect the activity and start pursuing the intruder once the item is picked up (assuming CCTV). If the intruder makes it to the door, it is considered a success for the intruder. Sigma has deliberately been denied direct programmatic access to the virtual environment here – as is provided in many VH and game applications – instead needing to perceive and act on the environment through a restricted, and deliberately noisy, perceptual-motor interface.

The original scenario involved only a single VH, for the intruder, but with part of the security functionality "grafted" into its brain; that is, a portion of it leveraged multiagent RL to learn a model of by which door the intruder would choose to exit (Ustun and Rosenbloom, 2015). This has since been extended to distinct intruder and security virtual humans. The intruder does not know the layout of the store and hence must learn a map and be able to use it to localize itself in the store. When the intruder locates the item of interest, it grabs the item and leaves the store via one of the exits. In the hypothetical retail store used here (Figure 84), there are shelves (gray rectangles), the item of interest (the blue circle) and two entry/exit doors (red rectangles). The intruder leaves the store via either the door by which it entered or the door closest to the item of interest. The main task for the security agent is to learn about the exit strategies of intruders and to use this knowledge to effectively detain them. However, the security agent also doesn't begin with knowledge about the layout of the store, and so must also learn a map.

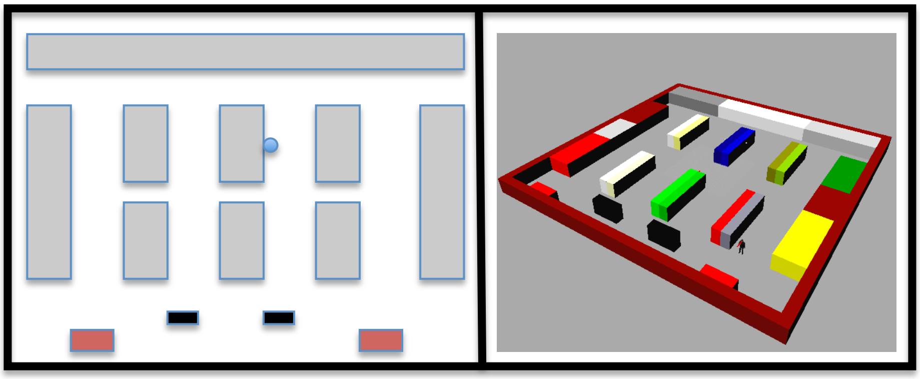


Figure 84: Layout of the store and its SmartBody representation.

SmartBody (Shapiro, 2011), a Behavior Markup Language (BML) (Kopp et al., 2006) realization engine, is used as the character animation platform, with communication between the Sigma VHs and SmartBody handled via BML messages. In the current setup, locomotion and path finding are delegated to the SmartBody engine. Sigma sends commands and queries to SmartBody to perform these tasks and to return perceptual information. Two basic types of perception are utilized by each of the two Sigma virtual humans in service of map learning: information about its current location, mimicking the combination of direction, speed and odometry measurements available to a robot; and objects that are in its visual field, along with their relative distances, mimicking the perception of the environment by a robot. Location information is conveyed to the Sigma VHs with noise added; perfect location information is not available to them.

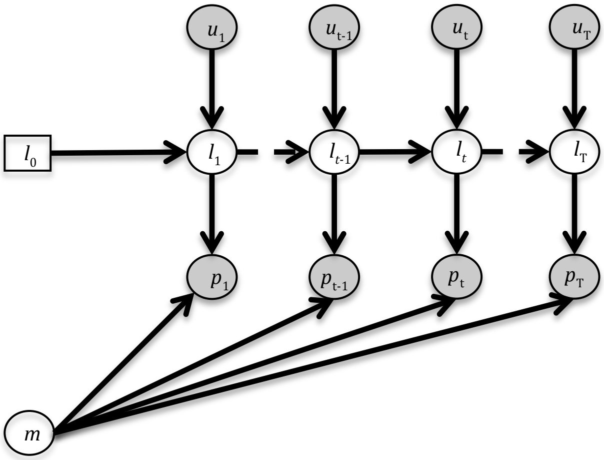


Figure 85: Dynamic Bayesian Network for SLAM.

A 31×31 grid is imposed on the store for map learning. A virtual human only occupies a single grid cell, whereas objects in the environment, such as shelves, can span multiple cells. The cognitive idiom for SLAM here, as shown in Figure 85, is a bit different from that used in the rest of this article, as a Dynamic Bayesian Network (DBN) (Murphy, 2002) is utilized in which l is the location, u captures the odometry readings, p represents perceptions of the environment, and m is the map of the environment. Locations are defined in terms of two predicates – `Location-X(x:location)` and `Location-Y(y:location)` – where the `location` type is discrete numeric with a span of 31. These are perceptual predicates, and hence induced perceptual buffers, that emulate odometry readings. In particular, perception of the current location of the agent involves a default noise model in which any cell adjacent to the correct one may be perceived as the agent's current location. In addition to the odometry readings, the objects in the visual field are also perceived, along with the relative distances along *x* and *y* of the centers of these objects to the agent.

The DBN is encoded via two almost identical conditionals, one for `Location-X` (Figure 86) and a similar one for `Location-Y`. These conditionals convert the perceived relative locations of the objects given the agent location to absolute locations in the map, using the affine transform *translation* to offset the agent's current location by the distances to the objects. In Figure 86, `Object-Location-X` is a memorial predicate whose LTMFN learns the *x* component of the map via gradient descent. Since both the `Location-X` and `Object-Location-X` patterns are condacts, the processing is bidirectional between them: both (noisy) perception of the VH's location and perception of the visible objects' relative locations have an impact on the posterior for the VH's location. This bidirectional processing forms the basis for SLAM, where the map is learned while it is simultaneously used for localization.

```
CONDITIONAL SLAM-X
   Conditions: Observed(object:o visible:true)
               Object-Distance-X(dist-x:dx object:o)
   Condacts: Location-X(x:lx)
             Object-Location-X(object:o x:(lx-dx))
```

Figure 86: SLAM conditional for the *x* coordinate.

The objective of the intruder is to grab the item of interest and to leave the store through one of the exits without being detained. In the current implementation, four basic behaviors are available to the intruder via operators that can be selected: (1) *walk towards target object*; (2) *run towards target object*; (3) *pick up target object*; and (4) *walk towards random object*. Target objects are important for the intruder to achieve its goals, while random objects drive exploration of the store.

The intruder is initialized with a sequence of target objects that it needs either to walk towards or to pick up. Given that the intruder has no a priori knowledge of the store, the location of the current target object may be unknown to it. In this case it needs to explore the store, mapping it in the process, to locate the target object. The basic operator used for exploration is *walk towards random object*. The intent is that doing this will help the agent discover new objects, and eventually the target object. This exploratory operator is always available; however, if other more task-relevant behaviors are available they take precedence. For example, Figure 87 shows a conditional in which the operator *walk towards target object* is proposed for selection, with a utility of 0.5, when the virtual human has seen the target object, and hence there is an estimate of its location. Exploration has a lower utility, so walking to a target object takes precedence if both operators have been proposed. When the virtual human is within a threshold distance of the target object, a new operator, *pick up target object*, is selected. The trial terminates when the intruder reaches its preferred exit door, which acts as the target object for the *run towards target object* operator.

```
CONDITIONAL Walk-Towards-Target
   Conditions: Target-Object(object:o)
               Seen-Objects(object:o visible:true)
    Actions: Selected(operator:walk-target)
    Function: 0.5
```

Figure 87: Conditional that proposes *walk towards target object*.

The security VH learns a map in the same manner as the intruder, but its primary task is to catch the shoplifter. One basic assumption here is that it is easy to recognize that a grab-and-run scenario has been initiated, by observing the pick-up behavior of the intruder. However, even though the security VH can thus easily recognize when such a scenario has been initiated, it still needs to intercept the intruder before it leaves the retail store. As there are two exit doors, early anticipation of the intruder's choice increases the chances of a successful detention.

In decision theoretic approaches to Theory of Mind, such as are leveraged here, agent models can be represented as reward functions. For the intruder in our scenario there are two possible models, distinguished by whether a reward is received when the agent returns to its door of entry or when it reaches the nearest door to the item of interest. Here, as in Pynadath et al. (2014), multiagent RL is leveraged in selecting among models of other agents; in particular, it enables the security VH to select a model of the intruder. First, RL is used to learn a distinct policy, or Q function, for the intruder under each possible model. These policies are then used in combination with the perception of the intruder's actions to determine which model yields a higher Q value for the actions actually performed by the intruder, and thus which model is more likely driving its action choices. This quickly enables the security VH to determine the correct door, with the Q value learned by RL substituting for what would otherwise be a need to explicitly extrapolate and compare the paths the intruder might take to the two doors.

As in Pynadath et al. (2014), it is possible to move from single-agent to multiagent RL, and from RL given a single reward function to RL given a range of possible reward functions, or models, by appropriately changing the predicates and conditionals. The conditional that compares the Q values and generates a distribution over the models is shown in Figure 88. It uses a filter to multiply the Q values for the observed action – specified by the location of the intruder and the direction of movement from that location – in each policy by 0.1, to scale utilities in [0,10] to values for selection in [0,1], and then projects these values onto the model predicate to generate a distribution over the model currently being used by the intruder.

```
CONDITIONAL Predict-Model
   Conditions: Previous-RL-Loc(location:loc)
               RL-Direction(direction:d)
               Q(model:m location:loc direction:d value:[0.1*q])
   Actions: Model(model:m)
```

Figure 88: Model prediction conditional.

In addition to demonstrating key – albeit still very initial – steps towards grand unification and generic cognition, the intruder and security virtual humans leverage Sigma's functionally elegant approach to implementing and integrating a range of both architectural and idiomatic capabilities. It also stretches, but does not quite break, the desideratum of sufficient efficiency. Although at ~250 ms the cognitive cycles were longer than the desired 50 ms, the impact on the performance of the VHs was minimal because those activities requiring faster decisions were delegated to SmartBody's internal algorithms (Ustun and Rosenbloom, 2015). Ultimately the goal is to move these activities fully into Sigma. There are also many additional capabilities that need to be, and can be, incorporated into these virtual humans, and many ways that this shoplifting scenario can be extended to provide new challenges.

8. Discussion

An evaluation of a long-term, large-scale research project – as opposed to a more traditional but narrower, scientific contribution – should focus on the scientific value of its desiderata, and how well these desiderata have been met. For Sigma, the case for the first part of this has been made in some detail in Section 2. Ultimately, the second part calls for challenging global evaluations, such as some variant of the Turing test or other comprehensive measures of human-level performance or human likeness. The virtual humans described in Section 7 are clearly initial steps in this direction, although they are still far from where they eventually need to be. In this section, an interim approach is taken to the second part that focuses on an informal analysis of where Sigma currently sits with respect to each of its four desiderata. This analysis will also act as a summary of the work described in this article, while pointing out gaps where future work is needed.

8.1 Grand Unification

Grand unification requires implementation of, and integration across, the full set of capabilities necessary for spanning the time scales – and associated levels and bands – shown in Table 1. Here we'll consider each of the four bands explicitly, beginning with the biological band, the natural home of work on neural/brain architectures and models, including recent attempts at whole brain models such as Blue Brain (Frackowiak and Markram, 2015) and Spaun (Eliasmith, 2013). In this article this band has been generalized to the more generic notion of a graphical band, which has then been modeled via Sigma's graphical architecture. The key motivation for beginning work on Sigma was the ability of graphical models to provide state-of-the-art algorithms across signal, probability and symbol processing from a single representation and processing algorithm, and even subsume several classical approaches to modeling neural networks. We have just begun to explore the relationship of Sigma's graphical architecture to neural networks, with a small extension for special one-way factor nodes that compute logistic functions, enabling the representation and solution of standard feedforward neural networks, although not yet with learning (Rosenbloom et al., 2016). But the bigger point is that results with Sigma to date have shown that the primary promise of graphical models for cognitive architectures/systems is in fact realizable. Signal processing shows up in vision (via a CRF) and speech (via an HMM); probabilistic reasoning appears in many places, but most obviously in semantic memory; and symbolic processing shows up in many places, such as in rule memory and problem-space search. Learning has also been demonstrated in these graphs.

The biggest current gaps from this perspective on the graphical band concern signal processing and learning. With respect to signal processing, piecewise-linear functions have turned out to provide a somewhat unwieldy means of representing, reasoning and learning with continuous functions; and learning so far has only yielded piecewise-constant functions that are even less amenable, particularly when spiky, for use in representing continuous functions. Both of these limitations point to a potential need to generalize further the function representation and learning algorithm for continuous signals. With respect to learning, beyond the current limitation to piecewise-constant functions, there is a general lack of reasonable learning for undirected functions – that is, ones with no child variables – and a complete lack of structure learning for acquiring and modifying nodes and links in the graph. Potential approaches to each of these gaps in Sigma can be imagined – for example, we are exploring a generalization of Soar's chunking mechanism to compile new graphical models from experience in reflective problem solving – but none has yet made its way into reality. There are also more minor issues with the need for alternative parameter values in gradient-descent learning, such as has been seen for both the learning rate and form of normalization used in episodic learning.

A second useful perspective on the graphical band is that it is the home to any necessary subcognitive processing. Beyond signal and probability processing, the recent work on appraisal and attention contributes here, but more work is required on additional appraisal variables and on further exploration of the implications of the attention mechanism. The rest of the emotional arc, in particular additional ways in which emotion may impact thought and behavior via coping and other mechanisms, also requires more work. Also missing here are motivation and personality (Sun and Wilson, 2010; Bach, 2015), and the notion of bodily state. With respect to motivation, we have begun exploring whether the application of desirability appraisals to variables representing basic needs might yield an approach akin to that embodied in MicroPsi 2 (Bach, 2015). With respect to bodily state, there is recent work, for example, that connects ACT-R with a state-of-the-art model of human physiology (Dancy et al., 2012), but from a generic cognition perspective, it would be useful to start from a more generic notion of bodily state, and of how such a state interacts with cognition, than is specifically provided by human physiology. The architectural appraisals implemented so far within Sigma start to approach this, but still leave a large gap to be filled.

The cognitive band is the natural home for work on traditional cognitive architectures, such as ACT-R and Soar. Sigma's cognitive architecture also sits within the cognitive band, as do its cognitive idioms. The many relevant varieties of memory, reasoning and learning that have been demonstrated within this band reveal the strength of Sigma here. There are almost certainly gaps, but they are smaller and thus less obvious. For example, there has not been much emphasis on planning in Sigma, but much that is known about planning in Soar (Laird and Rosenbloom, 1990; Rosenbloom et al., 1990) should transfer straightforwardly to Sigma. Two other potential gaps of note concern abductive reasoning, as is central in the Icarus architecture (Bridewell and Langley, 2011), and analogical reasoning, as is central in the Companions architecture (Forbus and Hinrichs, 2006). The former is hypothesized to play a significant role in understanding, for both natural languages in particular and for the world more generally. The latter has been hypothesized to play a critical role in much of cognition, and in particular for transfer of what has been learned to new situations.

Abduction has not yet been investigated in Sigma, but there has been some work on analogy. A simple form of analogy arises directly from leveraging the form of word/concept similarity that is implicitly represented in distributed vectors (Ustun et al., 2014). There has also been a preliminary investigation into how the more sophisticated approach to analogy that is embodied by the Structure Mapping Engine (Falkenhainer et al., 1989) might fit within Sigma, but there are not yet publishable results to report from this work. One hypothesis that has arisen from the combination of these two preliminary investigations is that distributed vectors might provide an appropriate reactive form of analogy, whereas an approach like SME might yield appropriate forms that are deliberative or reflective. Still, despite such potential gaps, the cognitive band is generally in good shape in Sigma.

The rational band is where (hierarchies of) goals combine with knowledge to yield rational responses to situations. It is the natural home for integrated systems focused more on the content of intelligent systems than on their architecture, such as Cyc (Lenat and Guha, 1990) and Polyscheme (Cassimatis, 2002). This is not yet a particular strength in Sigma, but some critical support for it does exist in terms of the availability of a blend of symbolic and probabilistic reasoning within the reactive control level; the existence of deliberative and reflective control levels that allow the system's full span of knowledge and processing to be brought to bear in service of achieving goals; and the recent addition of an architecturally penetrable representation for goals that enables the architecture to more directly contribute to rational goal-oriented behavior. If Sigma were to incorporate logics, they would be at this level, above the cognitive architecture rather than within it – but logics aren't currently included in Sigma. Whether or not this lack reflects a gap needing to be filled is open to judgment concerning the ultimate relevance of logic in natural and artificial intelligence. The biggest gap in the rational band is simply the absence of large bodies of diverse knowledge that can be leveraged in achieving goals. This may include abstract conceptual knowledge in the form of ontologies, laws and facts about the world (both qualitative and quantitative), knowledge of self, and more.

Newell (1990) actually said very little about the social band when he introduced the notion, but roughly it concerns organisms behaving appropriately in the context of other individuals and groups of individuals. It is the natural home of multiagent architectures including explicitly social architectures such as PsychSim (Pynadath and Marsella, 2005), while also depending critically on the ability to communicate – often in natural language – among the participants. The multiagent capability of Sigma, which enables multiple independent Sigma-based entities to run concurrently, and the associated work on Theory of Mind are initial attempts at approaching this band. The work on speech and language is also an important aspect of this as well. However, there are a number of major gaps that must be addressed before Sigma can be considered a fully social system. These include, but are not necessarily limited to a better understanding of other individuals plus any understanding of groups; inclusion of affective aspects of participating in social organizations; behaving appropriately in complex organizations with mixtures of shared and opposing goals; and full, bidirectional verbal and non-verbal communication.

Some cognitive architectures/systems have largely been initiated within a single band, such as Soar within the cognitive band and Spaun within the graphical band. Others have originated in a more explicitly cross-band manner, such as Clarion (Sun, 2006) and Sigma with origins that span the graphical and cognitive bands. Either way, if the development process continues for a sufficient period of time, attempts will almost inevitably be made to expand to other bands. Sigma is thus far from unique in this attempt, but it is one of the most well developed combinations of the graphical and cognitive bands – and one that takes a significantly different approach from other such attempts, in using graphical models in the graphical band in conjunction with an implementation relationship to the cognitive band – with also significant beginnings on both the rational and social bands.

In addition to spanning Table 1 in an integrated manner, grand unification also requires integration across the full arc of intelligent behavior, from perception and understanding, up through reasoning and cognition, and back down to planning and action, essentially covering all non-physical aspects of intelligent behavior within the system itself. Sigma has made a good start on this, with significant integration across central cognition, and with the cognitive cycle having been generalized to handle not just central cognition but also perception and understanding, including aspects of vision, speech and language. Speech processing – in addition to currently being limited to speaker-dependent, isolated-word, small-vocabulary tasks – is still fairly high level, beginning with acoustic labels rather than raw speech, and thus leaving a gap between the raw signal and these labels that currently is filled by external preprocessing. It is thus important to keep pushing Sigma further out towards the raw signal, not just in speech, but also in vision and in the perception of virtual worlds. There is even a bigger gap in motor control, where essentially nothing has been done to date, and a smaller gap in planning, where what has been learned about planning in Soar has mostly not yet been transferred to Sigma.

8.2 Generic Cognition

Generic cognition requires spanning both natural and artificial cognition at a suitable level of abstraction. At the bottom of the hierarchy in Table 1, the biological band has been abstracted to what was termed the graphical band, and then an implementation of this graphical band has been provided via graphical models. Graphical models are rooted in artificial intelligence and other engineering disciplines; however, they do generically model the parallel networks of local processing that are found in the brain and, as has already been mentioned, subsume several specific neural network models. As has also already been mentioned, gradient descent in factor graphs bears a generic resemblance to backpropagation in neural networks, and extensions to factor graphs are beginning to be explored that yield feedforward neural networks.

At the cognitive band, the cognitive architecture includes the distinction from cognitive science among working memory, long-term memory, and perceptual buffers. Via idioms, Sigma also reflects the further distinctions between procedural, semantic, episodic and imagery memories. There is even a temporal prior in episodic memory that maps onto the notion of base-level activation in ACT-R. In contrast, there are also CRFs, HMMs, SLAM, POMDPs, and RL from the artificial side, although RL has itself become successful of late as a model of learning in neuroscience (Niv, 2009). Sigma also embodies the tri-level control hierarchy inherited from Soar, which maps onto tri-level theories in both robotics and emotion, as well as onto the automated (System 1) versus controlled (System 2) distinction in cognitive science.

At the rational band, Sigma has stayed away from formal logics, based largely on the lead author's sense that not only are they not appropriate in cognitive models, but that they aren't even the best choice for building artificial systems. At the social band, Sigma has made use of concepts familiar from artificial intelligence such as Nash equilibria, and decision theoretic approaches are seen, but the approach is in fact based on PsychSim (Pynadath and Marsella, 2005), which is itself well grounded in the modeling of both small and large social systems.

In summary on generic cognition, it is clear that Sigma mixes ideas and constraints from both artificial intelligence (& robotics) and cognitive science, but it is not yet completely clear whether this means that it will yield both effective artificial systems and sufficiently accurate models of natural systems. Just to consider one major example from the natural side, does idiomatic memory, reasoning and learning only model human cognition at an abstract functional level, or does it reflect some deeper commonality waiting to be discovered in the structure of human minds and brains? Much of this more challenging aspect of generic cognition remains for future work.

8.3 Functional Elegance

Functional elegance requires broad cognitive (and non-cognitive) functionality – ultimately grand-unified generic cognition – from a simple and theoretically elegant base. There are currently three levels at which functional elegance can be assessed in Sigma: (1) from the graphical architecture to the cognitive architecture; (2) from the cognitive architecture to the cognitive idioms; and (3) from the cognitive idioms to virtual humans. The first and second levels have been covered in detail in Sections 5.4 and 6, respectively. Rather than reviewing these two levels here, we'll just state that they both exhibit extensive functional elegance, though not without a variety of rough edges, and leave to each reader to judge for her or himself those aspects that support the former versus exhibit the latter. The third level is just beginning to appear, and is thus too nascent to support much of an assessment. Still, even the early work in Section 7 is starting to reveal how combinations of cognitive idioms may lead to useful virtual humans, and how different combinations of such idioms may yield distinct varieties of virtual humans.

As noted in Section 2, functional elegance resembles the more standard scientific notion of *emergence*; however, it is a weaker notion that allows for some, although preferably minimal, explicit additions. Many of the distinctions in the graphical and cognitive architecture were introduced specifically to support the range of behavior that was to be produced above them, so it can't be said that all of these higher-level behaviors simply emerged. Nonetheless, they all are based on combinations of small numbers of what can be called *architectural microvariations*, rather than on distinct modules, thus fully complying with the notion of functional elegance.

8.4 Sufficient Efficiency

Sufficient efficiency requires executing quickly enough for real-time applications involving virtual humans and intelligent agents/robots, and for large-scale experiments in modeling human cognition. Given the optimizations described in Rosenbloom (2012c) and Rosenbloom, Demski and Ustun (2015), simple tasks typically run well within 50 msec per cycle, but more complex tasks can run considerably slower. The key determiner of combinatorics in graphical models is their *treewidth*, which in Sigma depends on the maximum number of dimensions/variables in functions and messages. However, the actual cost of Sigma's factor-graph operations depends on the *effective sizes* of its functions and messages, in terms of the number of distinct, non-zero regions they possess. In the worst case, the effective size of a function/message is the product of the domain sizes of all of its variables/dimensions, where the domain size of a continuous variable is effectively its span divided by the *epsilon* (i.e., 10^{-7} for an absolute comparison) that determines when two continuous values differ. Functional uniformity across this domain can, however, reduce the number of regions, and any empty regions – that is, ones with a functional value of 0 – can largely be ignored during processing. These two optimizations loosely correspond to leveraging *parameter equality* and *determinism*, respectively, in more traditional approaches to graphical models.

As was mentioned in Section 7, the speed of ~250 msec/cycle exhibited by the virtual humans in the shoplifting scenario was sufficient because behavior that needed to be controlled at shorter time scales was delegated to the body simulation system. As Sigma's coverage is pushed out more towards the periphery of perceptuomotor behavior, and to further subcognitive capability in general, the ~50 msec/cycle constraint will inherently tighten. At the same time, as the virtual humans and other systems developed within Sigma become more ambitious, and as learning is used more pervasively, the computational demands within the cognitive cycle will increase.

It may not be obvious a priori why parameter learning by itself might lead to increased costs, but episodic learning provides a simple example, where each cycle adds a new region in each episodic LTMFN whenever its value changes from the previous cycle. Thus such functions grow roughly linearly with time. Perhaps less obviously, functions such as those learned for the Q predicate in reinforcement learning also grow over time, possibly even exponentially. These functions are initialized to uniform distributions with just one region, but learning differentiates them, as necessary, as distinct subfunctions are learned for different combinations of states and operators. Consider a 2D function that is initialized with a uniform distribution, and thus only one region. Changing the function at just one point can add two slices along each dimension, yielding nine regions in the result. This growth is bounded by the worst case of the treewidth, but this can be a very large bound, particularly with continuous variables/dimensions.

Sufficient efficiency is thus a constant challenge in developing an architecture/system like Sigma. Optimizations over the past few years have gained two orders of magnitude in speed, but more is always necessary. Parallelism provides one obvious form of recourse that hasn't yet been explored, other than via a form of simulated parallelism in which messages along all link directions were effectively sent at once (Rosenbloom, 2012c). The search for better representations of graphical models and for faster algorithms usable in solving them must also continue unabated.

As mentioned in Section 2, in addition to the real-time aspect of sufficient efficiency that has so far been discussed in this section there is also a model-time component that concerns whether activities occur at human time scales when counting the number of primitive model steps, no matter how much real-time those model steps take. This relates both to how appropriately Sigma – particularly as partitioned into reactive, deliberative and reflective capabilities – maps onto the time scales in Table 1 in the context of grand unification, and to how well it handles the natural side of generic cognition. As already mentioned, the second part of this has largely been left for future work. With respect to the first part of it, Sigma's cognitive cycle is pegged to run at ~50 ms, providing a fixed model step for use higher in the hierarchy of layers and bands. A single cognitive cycle – or a simple reactive step – thus appropriately also maps onto a *Deliberate act*, at ~100 ms in the hierarchy, although the numbers are approximate enough that it could instead be two cognitive cycles, time enough to select and apply a non-problematic operator. The *Operations* layer in the hierarchy roughly corresponds to a minimal number of sequential/deliberative steps in service of behavior that requires more than simple reactivity. Exact mappings get fuzzy above this, and mappings below the *Deliberate act* are less meaningful because generic cognition is not concerned with precise modeling of networks of neurons.

9. Summary

At the highest level of description, what Sigma currently provides is a novel cognitive architecture and system that is distinctively characterized by the four desiderata introduced as the beginning of this article, with generic cognition and functional elegance the two that most distinguish it from other approaches and systems, and by the graphical architecture hypothesis. The specific contributions outlined in this article include the explicit explication of four key desiderata for consideration with respect to cognitive architectures/systems (Section 2); the two architectures embodied within Sigma, graphical and cognitive, and the functionally elegant implementation relationship between them (Sections 4 and 5); the cognitive idioms and how they are deconstructed in terms of microvariations in the cognitive architecture plus additional knowledge on top of it (Section 6); and the virtual humans plus how they are built from a combination of the cognitive architecture and idioms (Section 7). The discussion in Section 8 has provided its own contribution as well, yielding a first systematic – if still rather informal – analysis of Sigma with respect to its desiderata.

The many identified gaps that stand between where Sigma is today and where it ultimately needs to be lay out a rich array of topics for future work. In addressing these gaps we expect to continue leveraging the research strategy described in Section 1.2, which has stood us in good stead during the early years of this effort. Much of what is most needed at present in Sigma sits at the highest and lowest levels of analysis, such as further fleshing out of the social aspects of cognition; embodying large amounts of diverse knowledge; completing the subcognitive functionality needed for perception, motor control, emotion, and bodily state; and improving the implementation with respect to speed and scalability. Also critical is more integrative work on virtual humans, and possibly initial integrative work on intelligent robots and agents.

Acknowledgements

The work described in this article was sponsored by the U.S. Army, the Air Force Office of Scientific Research, and the Office of Naval Research. Statements and opinions expressed may not reflect the position or policy of the United States Government, and no official endorsement should be inferred. We would also like to thank our many collaborators on aspects of this effort, most of whom show up as authors or co-authors on one or more of the papers cited here.

References

Anderson, J. R. 1983. *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.

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

Anderson, J. R. 2002. Spanning seven orders of magnitude: A challenge for cognitive modeling. *Cognitive Science*. 26: 85-112.

Anderson, J. R. 2007. *How Can the Human Mind Occur in the Physical Universe*. Oxford: Oxford University Press.

Anderson, J. R.; Bothell, D.; Byrne, M. D.; Douglass, S., Lebiere, C.; and Qi, Y. 2004. An integrated theory of the mind. *Psychological Review*. 111: 1036-1060.

Badler, N. I. 1997. Real-time virtual humans. In *Proceedings of the IEEE Workshop on Non-Rigid and Articulated Motion*, 28-36.

Bach, J. 2015. Modeling motivation in MicroPsi 2. In *Proceedings of the 8th Conference on Artificial General Intelligence*, 3-13.

Bailey, T.; and Durrant-Whyte, H. 2006. Simultaneous localisation and mapping (SLAM): Part II State of the art. *Robotics and Automation Magazine*. 13: 108–117.

Bell, C. G.; and Newell, A. 1971. Computer Structures: Readings and Examples. New York, NY: McGraw-Hill.

Bengio, Y.; Ducharme, R.; Vincent, P.; and Janvin, C. 2003. A neural probabilistic language model. *The Journal of Machine Learning Research.* 3: 1137-1155.

Best, B.; Lebiere, C.; and Scarpinatto, C. 2002. A model of synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture. In *Proceedings of the Eleventh Conference on Computer Generated Forces and Behavior Representation*.

Bonasso, R. P.; Firby, R. J.; Gat, E.; Kortenkamp, D.; Miller, D. P.; and Slack, M. G. 1997. Experiences with an Architecture for Intelligent Reactive Agent. *Journal of Experimental and Theoretical Artificial Intelligence*. 9: 237-256.

Bostrom, N. 2001. Are you living in a computer simulation? *Philosophical Quarterly*. 53: 243-255.

Bridewell, W.; and Langley, P. 2011. A computational account of everyday abductive inference. In *Proceedings of the Thirty-Third Annual Meeting of the Cognitive Science Society*, 2289-2294.

Bubic, A.; von Cramon, D. Y.; and Schubotz, R. I. 2010. Prediction, cognition and the brain. *Frontiers in Human Neuroscience*. 4: 25.

Campbell, J.; Core, M.; Artstein, R.; Armstrong, L.; Hartholt, A.; Wilson, C.; Georgila, K.; Morbini, F.; Haynes, E.; Gomboc, D.; Birch, M.; Bobrow, J.; Chad Lane, H.; Gerten, J.; Leuski, A.; Traum, D.; Trimmer, M.; DiNinni, R.; Bosack, M.; Jones, T.; Clark, R.E.; and Yates, K.A., 2011. Developing INOTS to Support Interpersonal Skills Practice. In *Proceedings of the Thirty-second Annual IEEE Aerospace Conference*, 1-14.

Card, S. K.; Moran, T.P.; and Newell, A. 1983. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Cassimatis, N. 2002. *Polyscheme: A Cognitive Architecture for Integrating Multiple Representation and Inference Schemes*. Ph.D. diss., Media Laboratory, MIT, Cambridge, Mass.

Chater, N.; Oaksford, M. 1999. Ten years of the rational analysis of cognition. *Trends in Cognitive Sciences*. 3: 57–65.

Chen, J.; Demski, A.; Han, T.; Morency, L-P.; Pynadath, D.; Rafidi, N.; and Rosenbloom, P. S. 2011. Fusing symbolic and decision-theoretic problem solving + perception in a graphical cognitive architecture. In *Proceedings of the 2nd International Conference on Biologically Inspired Cognitive Architectures*, 64-72.

Collobert, R.; and Weston, J. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, 160-167.

Coste-Manière, E.; and Simmons, R. G. 2000. Architecture, the backbone of robotic systems. In P*roceedings of the International Conference on Robotics and Automation*, 67-72.

Dancy, C. L.; Ritter, F. E.; and Berry, K. 2012. Towards adding a physiological substrate to ACT-R. In *Proceedings of the 21st Conference on Behavior Representation in Modeling and Simulation,* 78-85.

Deering, S. 1988. *Watching the waist of the protocol hourglass*. Keynote address at ICNP '98.

de Kleer, J. 1986. An assumption-based TMS. *Artificial Intelligence*. 28:127–162.

Dempster, A.P.; Laird, N.M.; and Rubin, D.B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*. 39: 1–38.

Derbinsky, N.; Laird, J. E.; and Smith, B. 2010. Towards efficiently supporting large symbolic declarative memories. In *Proceedings of the 10th International Conference on Cognitive Modeling*, 49-54.

Deutsch, D. 2011. *The Beginning of Infinity: Explanations that Transform the World*. London, UK: Penguin Books.

Domingos, P.; and Lowd, D. 2009. *Markov Logic: An Interface Layer for Artificial Intelligence*. San Raphael, CA: Morgan & Claypool.

Douglass, S.; Ball, J.; & Rodgers, S. 2009. Large declarative memories in ACT-R. In *Proceedings of the 9th International Conference of Cognitive Modeling*, 222-227.

Eliasmith, C. 2013. *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford: Oxford University Press.

Falkenhainer, B; Forbus, K. D.; and Gentner, D 1989. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*. 41: 1–63.

Fikes, R., Hart, P; and Nilsson, N. 1972. Learning and Executing Generalized Robot Plans, *Artificial Intelligence*. 3: 251-288.

Forbus, K. D.; and Hinrichs, T. R. 2006. Companion Cognitive Systems: A step towards human-level AI. *AI Magazine*. 27:83-95.

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*. 19: 17-37.

Frackowiak R; and Markram H. 2015. The future of human cerebral cartography: a novel approach. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*. 370.

Frintrop, S.; Rome, E.; and Christensen, H.I. 2010: Computational visual attention systems and their cognitive foundation: A survey. *ACM Transactions on Applied Perception*. 7.

Garten, J.; Sagae, K.; Ustun, V.; and Dehghani, M. 2015. Combining distributed vector representations for words. In *Proceedings of the NAACL Workshop on Vector Space Modeling for NLP*, 95-101.

Goertzel, B. 2014. Artificial General Intelligence: Concept, State of the Art, and Future Prospects. *Journal of Artificial General Intelligence*. 5:1-46.

Goertzel, B.; Pennachin, C.; and Geisweiller, N. 2014. *Engineering General Intelligence*. Amsterdam: Atlantis Press.

Goodman, N. D.; Mansinghka, V. K.; Roy, D.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 220-229.

Hartholt, A.; Traum, D. Marsella, S. C.; Shapiro, A.; Stratou, G.; Leuski, A.; Morency, L.-P.; and Gratch, J. 2013. All together now: Introducing the Virtual Human Toolkit. In *Proceedings of the 13th International Conference on Intelligent Virtual Agents*, 368-381.

Hutter, M. 2005. *Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability*. Berlin: Springer-Verlag.

Itti, L.; and Baldi, P.F. 2006. Bayesian surprise attracts human attention. In *Advances in Neural Information Processing Systems 18*, 547-554.

Itti, L.; and Borji, A. 2013. State-of-the-art in visual attention modeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 35: 185-207.

Jilk, D. J.; Lebiere, C.; O'Reilly, R. C. and Anderson, J. R. 2008. SAL: An explicitly pluralistic cognitive architecture. *Journal of Experimental and Theoretical Artificial Intelligence*. 20: 197-218.

Jones, M. N.; and Mewhort, D. J. 2007. Representing word meaning and order information in a composite holographic lexicon. *Psychological review*. 114: 1-37.

Jordan, M. I.; and Sejnowski, T. J. 2001. *Graphical Models: Foundations of Neural Computation*. Cambridge, MA: MIT Press.

Joshi, H.; Rosenbloom, P. S.; and Ustun, V. 2014. Isolated word recognition in the Sigma cognitive architecture. *Biologically Inspired Cognitive Architectures*. 10: 1-9.

Kahneman, D. 2011. *Thinking Fast and Slow*. New York, NY: Farrar, Straus and Giroux.

Kieras, D. E.; and Meyer, D. E. 1997. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, 12: 391-438.

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

Kopp, S.; Krenn, B.; Marsella, S.; Marshall, A. N.; Pelachaud C.; Pirker, H.; and Vilhjálmsson, H. 2006. Towards a common framework for multimodal generation: The behavior markup language. In *Proceedings of the 6th International Conference on Intelligent Virtual Agents*, 205-217.

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

Laird, J. E. 2012. *The Soar Cognitive Architecture*. Cambridge, MA: MIT Press.

Laird, J E.; Newell, A.; and Rosenbloom, P. S. 1987. Soar: An Architecture for General Intelligence. *Artificial Intelligence*. 33: 1-64.

Laird, J. E.; and Rosenbloom, P. S. 1990. Integrating execution, planning, and learning in Soar for external environments, *Proceedings of the Eighth National Conference on Artificial Intelligence*,1022-1029.

Laird, J. E.; Rosenbloom, P. S.; and Newell, A. 1986. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*. 1: 11-46.

Langley, P.; and Choi, D. 2006. A unified cognitive architecture for physical agents. In *Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence*, 1469-1474.

Langley, P.; Laird, J. E.; and Rogers, S. 2009. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*. 10: 141-160.

Lebiere, C. 2013. Summary presentation for panel on Consensus and Outstanding Issues. *AAAI 2013 Fall Symposium on Integrated Cognition*.

LeCun, Y.; Bengio, Y.; and Hinton, G. E. 2015. Deep Learning. *Nature*. 521: 436-444.

Lenat, D.; and Guha, R. V. 1990. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Reading, MA: Addison-Wesley.

Madl, T.; and Franklin, S. 2012. A LIDA-based Model of the Attentional Blink. In *Proceedings of the 11th International Conference on Cognitive Modeling*, 283-288.

Maes, P.; and Nardi, D. eds. 1988. *Meta-Level Architectures and Reflection*. Amsterdam: North Holland.

Marr, D. 1982. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. San Francisco, CA: W. H. Freeman.

Marsella, S.; and Gratch, J. 2009. EMA: A Process Model of Appraisal Dynamics. *Journal of Cognitive Systems Research*. 10: 70-90.

McCallum, A.; Rohanemanesh, K.; Wick, M.; Schultz, K.; and Singh, S. 2008. FACTORIE: Efficient probabilistic programming via imperative declarations of structure, inference and learning. In *Proceedings of the NIPS workshop on Probabilistic Programming*.

Meyer, D. E.; and Kieras, D. E. 1997. A computational theory of executive control processes and human multiple-task performance: Part 1. Basic Mechanisms. *Psychological Review*. 104: 3-65.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations.*

Milch, B.; Marthi, B.; Russell, S.; Sontag, D.; Ong, D. L.; and Kolobov, A. 2007. BLOG: Probabilistic models with unknown objects. In *Introduction to Statistical Relational Learning* eds. L. Getoor and B. Taskar. Cambridge, MA: MIT Press.

Mnih, A.; and Kavukcuoglu, K. 2013. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in Neural Information Processing Systems*, 2265-2273.

Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon & Schuster.

Moors, A.; Ellsworth, P.C.; Scherer, K.R.; and Frijda, N.H. 2013. Appraisal theories of emotion: State of the art and future development. *Emotion Review*. 5: 119-124.

Murphy, K. 2002. *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. diss., Computer Science Division, UC Berkeley, Berkeley, Calif.

Murphy, R. R. 2000. *Introduction to AI Robotics*. Cambridge, MA: MIT Press.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.

Newell, A.; Shaw, J. C.; and Simon, H. A. 1959. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing,* 256-264.

Newell, A.; Yost, G. R.; Laird, J. E.; Rosenbloom, P. S.; and Altmann, E. 1991. Formulating the problem space computational model. In *CMU Computer Science: A 25th Anniversary Commemorative* ed. R. F. Rashid. New York, NY: ACM Press/Addison-Wesley.

Ng, A. Y.; and Russell, S. J. 2000. Algorithms for inverse reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, 663–670.

Niv, Y. 2009. Reinforcement learning in the brain. *The Journal of Mathematical Psychology*. 53: 139-154.

Oaksford, M.; and Chater, N. 2007. *Bayesian Rationality: The Probabilistic Approach to Human Reasoning*. Oxford: Oxford University Press.

O'Connor, T.; and Wong, H. Y. 2015. Emergent properties. In *The Stanford Encyclopedia of Philosophy (Summer 2015 Edition)*, ed. E. N. Zalta.

O'Reilly, R. C.; and Munakata, Y. 2000. *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. Cambridge, MA: MIT Press.

Ortony, A.; Norman, D. A.; and Revelle, W. 2005. Affect and Proto-affect in effective functioning. In *Who Needs Emotions? The Brain Meets the Machine* : eds. J. M. Fellous and M. A. Arbib. New York, NY: Oxford University Press.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufman.

Pynadath, D. V.; and Marsella, S. C. 2005. PsychSim: Modeling Theory of Mind with decision-theoretic agents. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 1181-1186.

Pynadath, D. V.; Rosenbloom, P. S.; and Marsella, S. C. 2014. Reinforcement learning for adaptive Theory of Mind in the Sigma cognitive architecture. In *Proceedings of the 7th Annual Conference on Artificial General Intelligence*, 143-154.

Pynadath, D. V.; Rosenbloom, P. S.; Marsella, S. C.; and Li, L. 2013. Modeling two-player games in the Sigma graphical cognitive architecture. In *Proceedings of the 6th Conference on Artificial General Intelligence*, 98-108. Berlin: Springer.

Rosenbloom, P. S. 1982. A world-championship-level Othello program. *Artificial Intelligence*. 19: 279-320.

Rosenbloom, P. S. 2006. A cognitive odyssey: From the power law of practice to a general learning mechanism and beyond. *Tutorials in Quantitative Methods for Psychology*. 2: 43-51.

Rosenbloom, P. S. 2009. Towards a new cognitive hourglass: Uniform implementation of cognitive architecture via factor graphs. In *Proceedings of the 9th International Conference on Cognitive Modeling*, 116-121.

Rosenbloom, P. S. 2010. Combining procedural and declarative knowledge in a graphical architecture. In *Proceedings of the 10th International Conference on Cognitive Modeling*, 205-210.

Rosenbloom, P. S. 2011a. Rethinking cognitive architecture via graphical models. *Cognitive Systems Research*. 12: 198-209.

Rosenbloom, P. S. 2011b. Mental imagery in a graphical cognitive architecture. In *Proceedings of the 2nd International Conference on Biologically Inspired Cognitive Architectures*, 314-323.

Rosenbloom, P. S. 2011c. From memory to problem solving: Mechanism reuse in a graphical cognitive architecture. In *Proceedings of the 4th Conference on Artificial General Intelligence*, 143-152.

Rosenbloom, P. S. 2012a. Deconstructing reinforcement learning in Sigma. In *Proceedings of the 5th Conference on Artificial General Intelligence*, 262-271. Berlin: Springer.

Rosenbloom, P. S. 2012b. Extending mental imagery in Sigma. In *Proceedings of the 5th Conference on Artificial General Intelligence*, 272-281.

Rosenbloom, P. S. 2012c. Towards a 50 msec cognitive cycle in a graphical architecture. In *Proceedings of the 11th International Conference on Cognitive Modeling*, 305-310.

Rosenbloom, P. S. 2014. Deconstructing episodic learning and memory in Sigma. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, 1317-1322.

Rosenbloom, P. S. 2015. Supraarchitectural capability integration: From Soar to Sigma. In *Proceedings of the 13th International Conference on Cognitive Modeling*, 67-68.

Rosenbloom, P. S.; Demski, A.; Han, T.; and Ustun, V. 2013. Learning via gradient descent in Sigma. In *Proceedings of the 12th International Conference on Cognitive Modeling*, 35-40.

Rosenbloom, P. S.; Demski, A.; and Ustun, V. 2015. Efficient message computation in Sigma's graphical architecture. *Biologically Inspired Cognitive Architectures*. 11: 1-9.

Rosenbloom, P. S.; Demski, A.; and Ustun, V. 2016. Rethinking Sigma's graphical architecture while extending it to neural networks. In *Proceedings of the 9th Conference on Artificial General Intelligence*.

Rosenbloom, P. S.; Gratch, J.; and Ustun, V. 2015. Towards emotion in Sigma: From Appraisal to Attention. In *Proceedings of the 8th Conference on Artificial General Intelligence*, 142-151.

Rosenbloom, P. S.; Laird, J. E.; and Newell, A. 1988. Meta-levels in Soar. In *Meta-Level Architectures and Reflection* eds. P. Maes and D. Nardi. Amsterdam, Netherlands: North Holland.

Rosenbloom, P. S.; Lee, S.; and Unruh, A. 1990. Responding to impasses in memory-driven behavior: A framework for planning, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 181-191.

Rosenbloom, P. S.; Laird, J. E.; and Newell, A. 1987. Knowledge level learning in Soar. In *Proceedings of Sixth National Conference on Artificial Intelligence*, 499-504.

Rosenbloom, P. S.; Laird, J. E.; and Newell, A. eds. 1993. *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA: MIT Press.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning representations by back-propagating errors. *Nature.* 323: 533-536.

Russell, S.; Binder, J.; Koller, D.; and Kanazawa, K. 1995. Local learning in probabilistic networks with hidden variables. In *Proceedings of the 14th International Joint Conference on AI*, 1146-1152.

Schneider, W.; and Shiffrin, R. M. 1977. Controlled and automatic human information processing: I. Detection, search, and attention. *Psychological Review*. 84: 1-66.

Shapiro, A. 2011. Building a character animation system. In *Proceedings of the 4th International Conference on Motion in Games*, 98-109.

Simon, H. A. 1956. Rational choice and the structure of the environment. *Psychological Review*. 63: 129–138.

Singla, P.; and Domingos, P. 2008. Lifted first-order belief propagation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 1094-1099.

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

Sun, R.; and Wilson, N. 2010. Motivational processes within the perception-action cycle. *Perception-Action Cycle: Models, Architectures, and Hardware*. New York, NY: Springer.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

Swartout, W. 2010. Lessons learned from virtual humans. *AI Magazine*. 31: 9-20.

Tambe, M.; and Rosenbloom, P. S. 1994. Investigating production system representations for non-combinatorial match. *Artificial Intelligence*. 68: 155-199.

Turney, P. D.; and Pantel, P. 2010. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*. 37: 141-188.

Ustun, V; and Rosenbloom, P. S. 2015. Towards adaptive, interactive virtual humans in Sigma. In *Proceedings of the 15th International Conference on Intelligent Virtual Agents*, 98-108.

Ustun, V.; Rosenbloom, P. S.; Kim, J.; and Li, L. 2015. Building high fidelity human behavior models in the Sigma cognitive architecture. In *Proceedings of the 2015 Winter Simulation Conference,* 3124-3125.

Ustun, V.; Rosenbloom, P. S.; Sagae, K.; and Demski, A. 2014. Distributed vector representations of words in the Sigma cognitive architecture. In *Proceedings of the 7th Annual Conference on Artificial General Intelligence*, 196-207.

Veloso, M. M.; and Carbonell, J. G. 1993. Derivational analogy in Prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*. 10: 249–278.

Veness, J.; Ng, K. S.; Hutter, M.; Uther, W.; Silver, D. 2011. A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*. 40: 95-142

Vere, S. and Bickmore, T. 1990. A Basic Agent. C*omputational Intelligence*. 6: 41-60.

Wang, P. 2007. The logic of intelligence. In *Artificial General Intelligence* eds. B. Goertzel and C. Pennachin. New York, NY: Springer.

Whiten, A. ed. 1991. *Natural Theories of Mind*. Oxford: Basil Blackwell.