Can Your Architecture Do This?
A Proposal for Impasse-Driven Asynchronous Memory Retrieval and Integration

Anthony G. Francis, Jr. and Ashwin Ram
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{centaur,ashwin}@cc.gatech.edu

Abstract

We propose an impasse-driven method for generating memory retrieval requests and integrating their contents dynamically and asynchronously into the current reasoning context of an agent. This method extends our previous theory of agent architecture, called *experience-based agency* (Ram & Francis 1996) by proposing a general method that can replace and augment task-specific mechanisms for generating memory retrievals and invoking integration mechanisms. As part of an overall agent architecture, this method has promise as a way to introduce in a principled way efficient high-level memory operations into systems based on reactive task-network decomposition.

Introduction

On the road to a theory of spontaneous memory retrieval in humans, we were struck by a revelation: the ability to respond to asynchronously generated information as it appears and to productively integrate it into one's thoughts and actions was of fundamental importance not only to humans, but to any intelligent, multifunctional agent living in a complex, dynamic environment. We took this principle and ran with it, developing a theory of agent architecture called *experience-based agency* based on parallel task execution, asynchronous memory retrieval, and dynamic integration mechanisms.

In the process of implementing and testing this theory in the context of a case-based planning system called Nicole-MPA, we have discovered a method to generate retrieval requests and respond to memory retrievals in a principled way; this discovery is the next steppingstone in our quest to transform the experience-based agency theory from a specification for the design of task-specific agents to a theory of cognitive architecture for general intelligent action.

Recalling the Perfect *Bon Mot*

Let's unpack this a bit. As an agent thinks about and acts in the world, its reasoning process is constantly generating, retrieving and processing information relevant to the task it is performing. But most of the information provided by the environment — and, in humans, a good bit of the information provided by our memories — often seems irrelevant to our current processing, or, as in the case of the exit sign that comes up too fast when driving or the perfect *bon mot* recalled too late in a conversation, seems to arrive on its own perverse schedule, rather than when we need it.

But an efficient agent can actually take advantage of this kind of asynchronous information; whether it comes from the outside world or the inside of one's head, an agent should have the capability to shift lanes quickly (when it is safe), or to steer the topic subtly backwards to deliver the atomic catchphrase (unless it has grown too stale). At a minimum, two essential capabilities are required: the ability to integrate asynchronous information into the current reasoning context, and the ability to judge whether or not it is productive to do so.

But in addition to these, humans have a third capability: the ability to internally generate — *to be reminded of* — information which might be relevant to current problem solving. To model this ability, we have developed an asynchronous memory retrieval algorithm based on the following principles. Given an uniform knowledge base which stores all an agent's experiences (called an "experience store") and a working memory which stores the agent's current reasoning context, asynchronous retrieval can be achieved by a memory task acting in

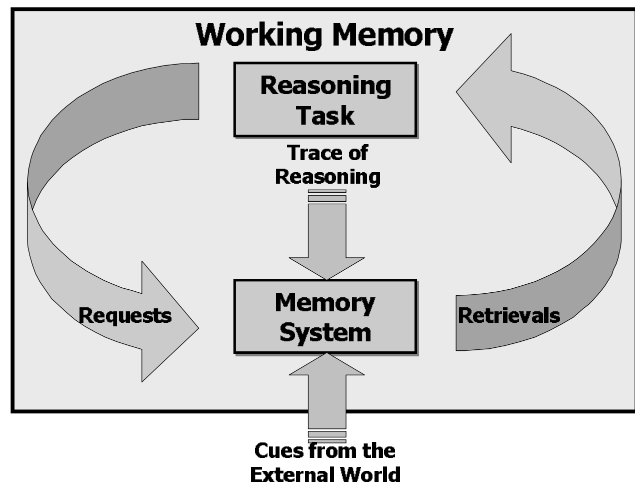

Figure 1. The core idea of asynchronous memory

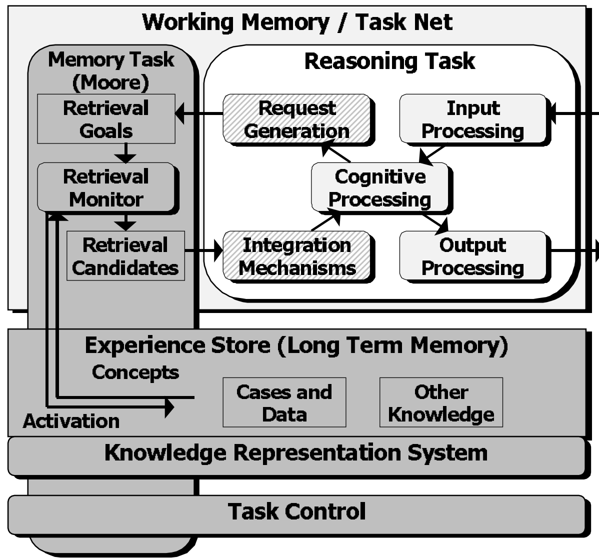

Figure 2. The Architecture of Nicole

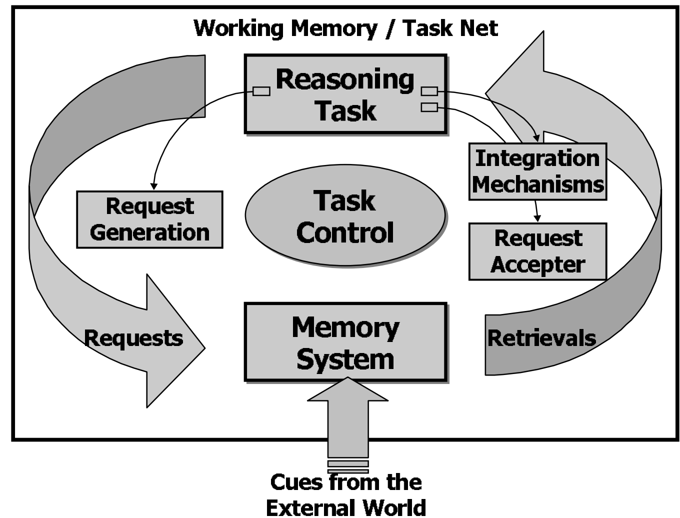

Figure 3. How experience-based agency elaborates the basic asynchronous memory model.

parallel with other reasoning tasks within the agent, searching the experience store on its own schedule and returning candidate retrieval items as soon as they are found. Memory's search is *guided* by whatever is visible in working memory (such as a reasoning trace or cues from the outside world) but it remains independent. The only *explicit* channels of communication between memory and reasoning are retrieval requests and retrieval responses, as Figure 1 illustrates. Appendix A provides more detail on our implementation of asynchronous memory.

Agent Specification, Agent Architecture

But asynchronous memory by itself does not an agent architecture make; to build an agent around an asynchronous memory we must specify further details, such as how retrievals are integrated into reasoning and how the agent controls its behavior. We have combined the constraints of the asynchronous memory theory (a uniform knowledge base or "experience store", a global working memory, and an asynchronous memory retrieval process) with two additional hypotheses (namely, that knowledge is integrated through dedicated, reasoning-task-specific subtasks called integration mechanisms, and that decisions about integration are executed by a global task control mechanism) into an overall theory called *experience-based agency* (Ram & Francis, 1996).

The experience-based agency theory does not commit to any particular reasoning method or problem solving strategy, and therefore is more properly termed a specification for agent design than a complete cognitive architecture for general intelligence. Figure 2 illustrates this distinction: the processes and data structures in grey are fully specified by the theory, whereas the details of the reasoning task depend on the design of the agent or system. Our usage of the theory backs this up; we have implemented the theory in an agent framework called Nicole, but our evaluation has focused more narrowly on a case-based planning system called Nicole-MPA.

One thing we discovered in the course of our development of Nicole-MPA is that the design of the task controller — and the types of task networks that it supported — critically determined the types of reasoning the system could perform. While it was *possible* to program a wide variety of behaviors in the initial task language, complex interleaving of memory and reasoning were difficult; furthermore, all memory requests, retrieval processing, and integration mechanisms had to be handled through highly explicit, reasoning-task dependent code. Figure 3 illustrates this elaboration of the original asynchronous memory model (small arrows indicate that one task is a subtask of another). As a professor of one of the authors once said, "Ain't nothing simple when you're doing it for real" (Baird 1989).

As we experimented with the implementation and discovered its limitations, we refined the theory behind it. While the basic theory behind the experience store, the working memory, the asynchronous memory system, and our specific choices of integration mechanisms have held up under the pressure, the theory behind task control has been considerably elaborated. While this elaboration does not "close the loop" and provide a complete specification for a complete cognitive architecture, it has provided us with a way to automatically generate certain memory requests, process the corresponding retrievals when they occur, and invoke the appropriate integration mechanisms when needed — the striped boxes in Figure 2.

Impasse-Driven Memory Retrieval

At the highest level, task control in the current experience-based agency theory can be described as recursive, reactive task decomposition, and shares a common heritage with systems such as RAPs (Firby 1989), TMK models (Goel & Murdock 1996) and generic tasks (Chandrasekaran 1989), to name a few. But it also shares some properties with systems ACT* (Anderson 1983) in that much task processing occurs through the operation of

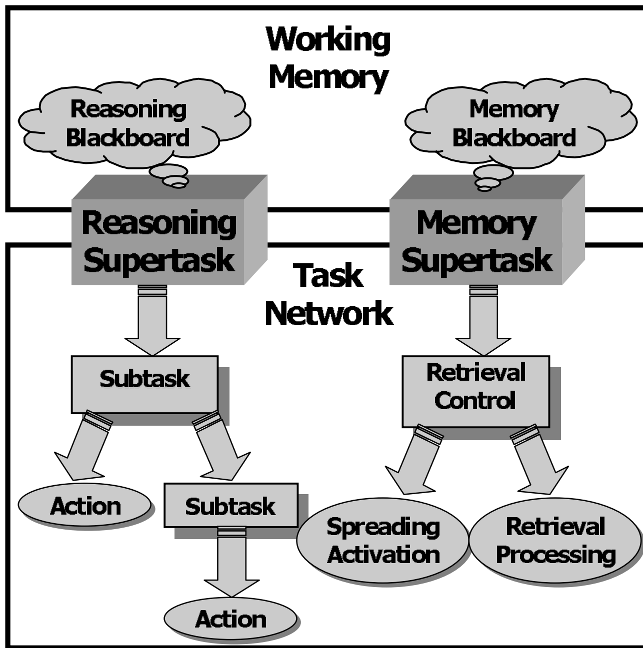

Figure 4. Task Decomposition in Nicole.

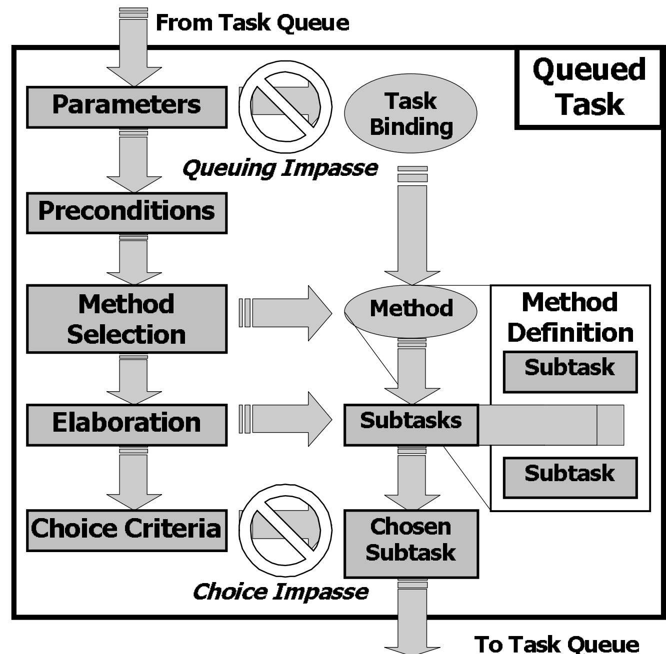

Figure 5. Potential Impasses in Task Decomposition.

productions and with systems like Soar (Newell 1990) in that task processing steps can impasse if appropriate knowledge is not available. Two classes of impasses — queuing impasses and choice impasses — provide opportunities to automate asynchronous memory retrieval.

Task Processing in EBA

The root of the task system consists of several *supertasks* — high-level objects, corresponding to major cognitive functions such as memory, reasoning and perception, which structure the system's working memory and spawn specific subtasks to achieve the cognitive functions (Moorman & Ram 1996, Ram & Francis 1996). Figure 4 illustrates supertasks' dual role as both memory and processing structures. By definition, all supertasks are processed in parallel; beneath that level, things get more complicated.

There are five steps to task processing in an experience-based agent: method selection, method elaboration, subtask choice, task queueing, and task application. Once a task has been queued for execution, a method must be selected to actually execute a task. A method, which may specify a complex network of subtasks acting serially or in parallel, must be elaborated to propose a set of subtasks for execution. Those tasks are evaluated and, depending on the task network structure, one or more are chosen for execution. If the chosen subtask's parameters can be bound and its preconditions satisfied, the subtask will be queued for execution, and finally will be applied — recursively decomposed if it is a complex task, or executed immediately if it is atomic.

Figure 5 illustrates the task decomposition process process from the perspective of a task waiting in the execution queue. First, its parameters are bound and precondition satisfied, then, a method is selected. That method is then elaborated to select a set of candidate subtasks, one or more of which may be chosen for execution.

Impasse-Driven Retrieval Request Generation

When no productions or atomic tasks exists to implement a task processing step, the task system can impasse. In particular, when a method specifies that a subtask must run to completion and that subtask's preconditions cannot be satisfied, the result is a *queuing impasse* (shown as the top impasse in Figure 5). One way to resolve a queuing impasse is to retrieve an item from memory; traditionally, this has been done in the implementation with reasoning-task specific code.

However, a task's preconditions and parameters can specify both the type and structure it needs, as well as where in the working memory that knowledge should appear. It just so happens that the specification of the type of memory needed for a task is almost identical to the core of a retrieval request to the memory system (although memory retrieval specifications can be further elaborated). Thus, when a queuing impasse occurs, a memory retrieval request can be automatically spawned and processed in parallel; once an item has been retrieved asynchronously, it can be posted to working memory, allowing the task to proceed.

While this method does not encompass high-level strategic memory requests, it does provide an automatic way to detect and satisfy a reasoning task's needs for information from long-term memory, something that previously had to be done by hand. Impasse-driven retrieval request generation is thus a weak method for knowledge retrieval — a general method for performing a task which formerly had to be accomplished through knowledge-intensive problem solving methods. Figure 6

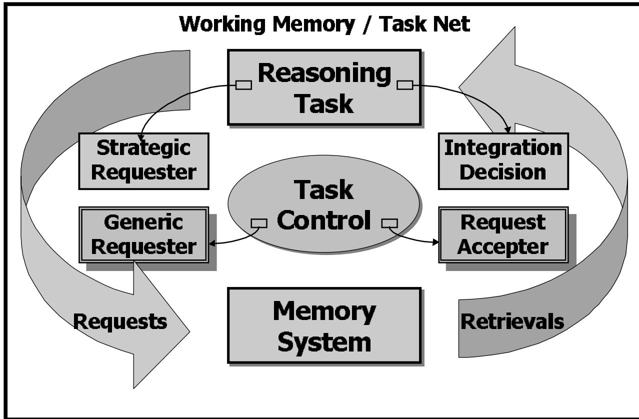

Figure 6. Impasse-Driven Retrieval Request Generation

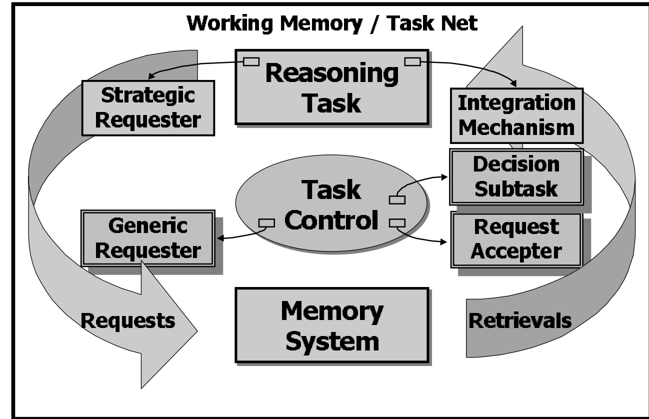

Figure 7. Impasse-Driven Integration Decisions

illustrates how this method elaborates our original picture of asynchronous memory retrieval by explicitly separating strategic and "normal" retrieval and by the addition of generic request mechanisms; note how the generic requester and the request accepter are now subtasks of the task control system, rather than the reasoning task.

Great. But this raises another question: once this information is retrieved, how can we be sure it will not disrupt the rest of the reasoning process — which may have been elaborating and executing other tasks in parallel?

Impasse-Driven Integration Mechanism Invocation

The solution to this dilemma again lies in how a task method is specified. Just as a method can specify that a task must run to completion, it can also specify that out of several alternative subtasks, only one may be chosen. When no information exists to choose a subtask, we have a *choice impasse* (shown as the bottom impasse in Figure 5).

Integration mechanisms encapsulate (implicitly or explicitly) three types of knowledge: how to prepare raw retrievals from memory to make them suitable for a particular reasoning context, how to actually merge the prepared retrievals into the current reasoning context, and evaluation metrics on when this retrieval is fruitful. While a great deal of preparation can be done in parallel to ongoing reasoning tasks, merging cannot; by definition merging represents a departure from the course of ongoing reasoning. This represents a natural choice point, and unless some information is available to discriminate between these choices a choice impasse will arise.

Note that since merging subtasks are by definition optional, no queuing impasses will arise if their preconditions cannot be satisified. Only if some other task satisfies their preconditions for them — such as a preparation task, which can run to completion in parallel and can hence generate memory retrievals — will they be candidates for queuing, and only then will a choice impasse arise. When the choice impasse does arise, the evaluation subtask of the integration mechanism can be invoked, allowing the system to decide whether to continue reasoning or to attempt to merge.

Because integration mechanisms are intimately tied to the reasoning processes, no completely general method can be devised to automatically perform integrations or to evaluate their utility. However, using choice impasses as the vehicle to orchestrate integration mechanisms gives us a clear account of the content needed in an integration mechanism, a structure for the storage of that content, and a uniform mechanism for its execution. Figure 7 illustrates this in action: even though a choice mechanism may be provided by a reasoning task, it is invoked automatically as a subtask of the task controller when a choice impasse occurs.

Why Should You Care?

Using explicit impasses to automatically generate memory retrieval requests and to organize the implementation of integration mechanisms certainly makes our lives easier, but why should the designers of intelligent agents care about these methods, especially if they aren't using an explicit instantiation of the experience-based agency theory? The answer is twofold: first, this method teaches a general lesson about intelligent agents in particular, and second, this method makes an agent built on the experience-based agency theory in particular a more viable choice for an implementation.

First, asynchronous memory retrievals coupled with integration mechanisms are powerful tools to simultaneously deal with both novel information from the environment and to enlist the an agent's own reasoning processes (through the working memory trace) in the task of effectively exploiting the agent's past experiences. But making asynchronous integration work requires a powerful controller and a methodology for choosing when to retrieve and when to integrate. Even if the actual implementation of a system diverges wildly from the type of task decomposition used in an experience-based agent, a task-method-knowledge analysis (TMK) of the reasoning process within an agent (Goel & Murdock 1996) can provide pointers of where to retrieve, where to prepare, where to merge, and where to choose to integrate.

Second, with the addition of this method, designers building systems explicitly based on experience-based agency principles no longer need to develop retrieval

request generators, retrieval acceptors, integrators and choice routines in an ad-hoc fashion; they are now a part of the architecture, determined by the interaction of memory and task control (and to a lesser extent by the interaction of task control and specific reasoning mechanisms).

The future of this research is also twofold. First, these extensions to the theory of experience-based agents need to be implemented and tested in the context of our testbed system, Nicole; this will no doubt expose new problems and, hopefully, new opportunities. Second, the shift from poorly-defined task packages and execution modules of the old theory to the explicit task decomposition with productions, preferences, and impasse-driven spawning of new tasks lays the foundation to developing general problem solving methods and a more complete cognitive architecture for general intelligent action.

Acknowledgements

This research was supported by the United States Air Force Laboratory Graduate Fellowship Program, by the Air Force Office of Scientific Research, and by the Georgia Institute of Technology.

References

Anderson, John R. (1983). *The Architecture of Cognition*. Cambridge, Massachusetts: Harvard University Press.

Baird, g. (1989). Personal communication.

Chandrasekaran, B. (1989). Generic tasks as building blocks of knowledge-based systems: the diagnosis and routine design examples. *Knowledge Engineering Review, 3(3):* 183-219, 1988.

Firby, J. (1989). Adaptive Execution in Complex Dynamic Worlds Ph.D. Thesis, Yale University Technical Report, YALEU/CSD/RR #672, January 1989.

Goel, A., & Murdock, W. (1996). Meta-cases: Explaining case-based reasoning. In Ian Smith & Boi Faltings, (eds.), *Advances in Case-Based Reasoning: Lecture Notes in Computer Science 1168*. Springer.

Klimesch, W. J. (1994). *The structure of long-term memory: A connectivity model of semantic processing*. LEA.

Moorman, K. & Ram, A. (1994). Integrating Creativity and Reading: A Functional Approach. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, Atlanta, GA, August 1994.

Newell, A. (1990) *Unified theories of cognition*. Harvard.

Ram, A., & Francis, A. G. (in press). Multi-Plan Retrieval and Adaptation in an Experience-Based Agent, to appear in D. B. Leake, editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, AAAI Press.

Desired Property	Properties of Agent	Properties of Memory
asynchrony	concurrent tasks, task controller	reified retrieval requests, retrieval monitor
context sensitivity	working memory	context-directed spreading activation

Figure 8. Properties of Experience-Based Agents and their Memories

Appendix A. Asynchronous Memory

So, how can we construct a memory system which operates independently from other reasoning tasks, yet remains sensitive to cues from reasoning and cues from the outside world — that is, a memory which is both asynchronous and context sensitive? Part of the answer, of course, depends on the construction of the agent: without some equivalent of Nicole's task controller to interleave tasks, asynchronous retrieval doesn't even make sense. But the larger part of the answer depends on the construction of the memory itself.

We propose that asynchrony can be achieved through the use of reified retrieval requests and a retrieval monitor which operate in conjunction with the agent's task controller, and that context sensitivity can be achieved through a process, called context-directed spreading activation, which operates hand in hand with the agent's working memory. Figure 8 illustrates how achieving asynchronous, context-sensitive retrieval depends upon the properties of both the memory and the agent; we believe this memory architecture could function equally well in any similarly equipped agent.

Asynchronous Retrieval Asynchronous retrieval in this architecture is achieved using *reified retrieval requests* managed by a *retrieval monitor*. Reified retrieval requests are first-class knowledge objects in the experience store that record all the information associated with a request for information from the memory system — the type of request, the specification of the item wanted, the asking task, and so on. These are essentially glorified knowledge goals, given a privileged position within an experience-based agent's architecture through the operation of the retrieval monitor.

The retrieval monitor keeps track of the requests made by reasoning tasks with a priority queue of retrieval requests, with the option to terminate or suspend low-priority requests if too many resources are being expended upon retrieval. Upon each *retrieval cycle*, the monitor compares the specifications in the retrieval request queue with most active items in the experience store (as determined by a *selection task*; see the following section), posting an *alert* to the working memory when complete.

In addition to the traditional features of retrieval — receiving requests, returning responses (alerts) — there are some novel features of this memory system tied in directly to its asynchronous nature. Reasoning tasks can demand a *best guess* initially and then allow the monitor to continue processing the request at a higher level of sensitivity, *updating* the request with new cues or specifications as needed. When an alert is posted, a task can *accept* or *reject* a candidate retrieval. Finally, tasks can either *cancel* or *accept* the work the memory system has done on a memory request — regardless of whether it ever returned any results. Because of these additional features of the retrieval process, reified retrieval requests contain

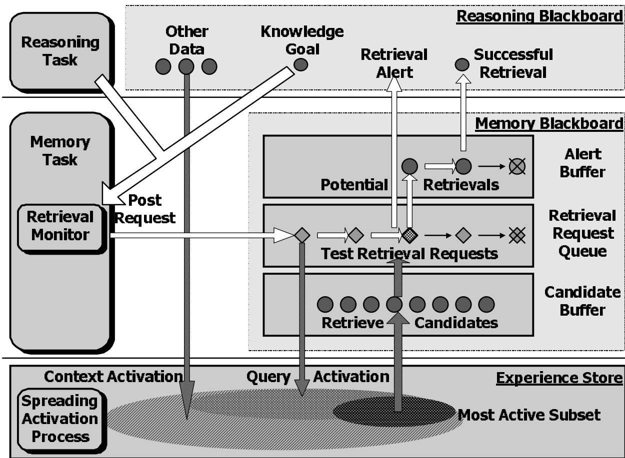

Figure 9. The Life History of a Retrieval in Nicole.

information beyond that necessary for traditional retrievals — lists of current candidates, histories of past accepts and rejects, priority levels, sensitivity levels, and so on. Figure 9 illustrates the life history of a retrieval request in this architecture (for simplicity, retrieval requesters, retrieval accepters, integration mechanisms and choice mechanisms are omitted from this diagram; of course, from the perspective of the innards of the memory module, these additional tasks are invisible).

Context Sensitivity By themselves, reified retrieval requests and a retrieval monitor could make up the core of an asynchronous memory system as long as *some* task existed to select "the most active items in the experience store." This could be as simple as a sequential search of memory items examining some fixed number on every cycle or as complex as hashed search based on the specifications provided to the memory module. However, we want a memory system for an experience-based agent to be efficient, which rules out sequential search of a experience store; and we want it sensitive to context, which probably rules out simple hashed search based on specifications alone.

We propose that *context-directed spreading activation* be used as the primary selection task. Context-directed spreading activation uses the set of currently active items in the memory system to direct and inform further activation, on the theory that memory requests are best served warm — that is, most memory requests can be satisfied with concepts closely related to the concepts that the agent has been thinking about or has encountered in its environment. In psychological terms, context-directed spreading activation is a priming or preactivation process (for an overview, see Klimesch 1994).

Thus, there are *two* sources of activation in an experience based agent: query activation, which spreads from the specific knowledge items that are part of retrieval requests, and context activation, which spreads from items stored in the system's working memory. Query activation is propagated explicitly whenever a retrieval request is created; context activation is propagated implicitly, through the add/delete hooks in the working memory. Other than their source, query activation and context activation are identical, exploiting the same context-directing spreading activation process.

Currently, we use two implementation mechanisms for context-directed spreading activation.

- *context activation*: activation spreads more efficiently to items which are already activated above some threshhold value.
- *gated spreading activation*: activation spreads more efficiently along links mediated by relation nodes which are active.

In more detail, the change in activation that propagates from a node *j* to a node *i* along a link mediated by relation node *r* is determined by the equation:

$$\underset{j \rightarrow i}{\Delta} a_i = \frac{\sum_{\left\langle r,j \,\middle|\, j \overset{r}{\Rightarrow} i \right\rangle} \left(P_{base} + P_{context} a_i\right)\left(R_{base} + R_{gating} a_r\right) S_{j \rightarrow i} a_j}{\sum_{\left\langle all\, r,k \,\middle|\, j \overset{r}{\Rightarrow} k \right\rangle} \underset{j \rightarrow k}{\Delta} a_i}$$

where:

$\mathbf{a_i}$	activation of node *i*
$\mathbf{S_{i \rightarrow j}}$	strength of the link between nodes *i* and *j*
$\mathbf{P_{base}}$	basic node propagation parameter
$\mathbf{P_{context}}$	active node bias parameter
$\mathbf{R_{base}}$	basic relation propagation parameter
$\mathbf{R_{gating}}$	active relation bias parameter

While this equation is complex,[1] its behavior is easy to understand if the various parameters are pushed to limiting values. The bias parameters serve to determine the degree of context activation. Raising the context bias parameter $\mathbf{P_{context}}$ for nodes increases the ease with which activation spreads to active nodes; raising the gating bias parameter for relations increases the ease with which activation spreads along links whose relations are active.

When the bias is set to zero, context-directed spreading activation devolves to traditional unbiased spreading activation with fan-out, such as proposed by Anderson (1983). The base parameters $\mathbf{P_{base}}$ and $\mathbf{R_{base}}$ determine the properties of this process. If the base parameters are set to zero, essentially the only activation that can propagate is context-based: activation can only spread to nodes with some existing degree of activation, and only along links whose relations are active. In practice, intermediate values are chosen for both the base and bias parameters, allowing both context-directed and traditional spreading activation.

[1] This equation is actually simplified; there are a number of additional parameters to the context-directed spreading activation process (such as the total number of propagating nodes and the degree of fan-out limitation on spreading activation) that are required by details in the experience-based agency theory which are beyond the scope of this paper.