Storm: A Framework for Biologically-Inspired Cognitive Architecture Research

Douglas Pearson (douglas.pearson@threepenny.net)
ThreePenny Software, 4649 Eastern Ave. N.
Seattle, WA 98103 USA

Nicholas A. Gorski (ngorski@umich.edu)
Richard L. Lewis (rickl@umich.edu)
John E. Laird (laird@umich.edu)
University of Michigan
Ann Arbor, MI 48109 USA

Abstract

We have developed a software framework called Storm to aid the development of cognitive architectures based on the structure and function of the brain. The goals of the framework are to make it both easy and fast to develop and experiment with alternative architectures and components of architectures. In addition, the framework supports explicitly mapping its components to structures in the brain. We demonstrate a working implementation of the framework, where we have developed a simple model of skill learning and memory management in a simple 2D grid world.

Introduction

In cognitive modeling, there is a divide between models that attempt to capture the details of neural activity and those that attempt to model complex overt behavior. Models of complex behavior often use combinations of symbolic and non-symbolic representations of knowledge in cognitive architectures. Detailed models at the neural level posit direct mappings to structures and processes of neural systems in the brain. The achievements of neural models to date have been impressive (Munakata & Johnson, 2006); but it is very difficult to create models of the interactions of sufficient brain systems for anything approaching a complex task (e.g., see Simen et al, 2004 for a recent attempt). Conversely, the cognitive architecture approach has been successful at modeling a wide variety of complex tasks (e.g., see the models reported in Gray, 2007) but the mapping of the components of those models to structures and processes in the brain often remains unclear – although Anderson's recent work has demonstrated that it is possible to map some structures in ACT-R to specific brain regions (Anderson, 2007).

We propose an alternative between high-level cognitive architectures and low-level neural models. Our approach is to create architectures composed of models of brain structures and their interconnections (at possibly multiple levels of abstraction) – a brain-based architecture capable of cognitive behavior.

In order to pursue this approach, it behooves us to take a step back and not jump immediately into the construction of a specific architecture. Instead, the first step, and the subject of this paper, is to develop a software framework in which such models can be easily developed, tested, evaluated, and extended. More specifically, we believe a useful framework will support:

(1) Rapid prototyping of architectures composed of a heterogeneous collection of interacting components operating in parallel, with their own possibly unique time scales, processes and representations.
(2) Easy and efficient simulation of the dynamics of such architectures.
(3) Explicit and flexible mappings of architecture-to-brain structure, and easy exploration of the implications of such mappings for predictions of brain activity. The framework should also make it easy to exploit existing detailed databases on brain structure and brain connectivity (Alexander, Arbid & Weitzenfeld, 1999).
(4) Maximal flexibility in programming languages, operating systems, and parallel computation.

Furthermore, our goal is not only to develop a tool to aid our own research, but a tool that others will use for their own explorations, thereby facilitating the sharing of components between research groups.

In this paper, we describe the Storm framework, a software infrastructure intended to realize the above goals. Using the initial implementation of Storm, we developed a simple architecture that includes action selection, reinforcement learning, and simple long-term and short-term memories. This architecture is not (yet) meant to be a faithful model of brain structures, but is meant to demonstrate the capabilities of Storm.

Various features and motivations for Storm have precedent in the cognitive modeling and neural network modeling communities. The explicit goals of Storm are perhaps most closely aligned with NSL (Neural Simulation Language; Weitzenfeld, Arbib & Alexander, 2002). There are key differences, however. NSL defines a new object-oriented language that must be used for creating models. In contrast, Storm allows users to develop architectural components in standard computer languages (C++, Java), while providing support for communication between modules and scheduling model execution. In addition, Storm provides facilities for explicitly mapping model components to brain structures. Storm also differs from neural network toolkit approaches such as Leabra (O'Reilly & Munakata, 2000) and Eliasmith and Anderson (2002) because it has no a priori bias to specific models of the brain or neurons. Finally, we note that the goal of providing an appropriate abstraction layer for building event-driven simulations is also adopted in the implementations of some existing cognitive architectures, including ACT-R (Bothell, 2004) and Epic (Kieras & Meyer, 1997), though neither of these architectures embraces the general framework goals described above.

The Storm Framework

Defining an architecture

The framework must have a way of representing architectural components, how they interact and the computations they perform. In the Storm framework, we decompose an architecture into two types of components: *function modules* and *state variables*. Function modules perform processing while state variables hold persistent structures and provide communication between function modules. This approach naturally reflects a simple dynamical systems view of brain architecture, in which the union of state variables represents the current state of the system, and the function modules represent the dynamic relationships among those state variables.

Function modules receive inputs from a set of state variables and generate outputs to one or more state variables. Figure 1 shows a function connectivity graph for a simple architecture with two function modules (the M1 & M2) and four state variables (the circles). In this case, the state variables A, B, and C are the inputs for functional module M1, which generates outputs for A and D. Module M2 receives input from D and generates output for C. The graph is not explicitly represented in the framework as a separate data structure, but is implicitly defined by the inputs and outputs of the modules.

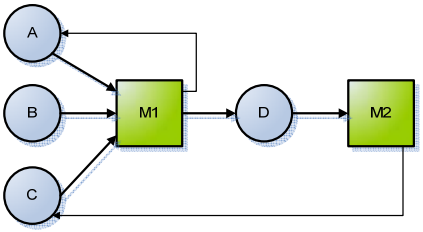

Figure 1: Functional Connectivity Graph

An architecture's decomposition into state variables and function modules represents theoretical commitments about brain architecture. It is possible to build both highly interactive and highly encapsulated systems and subsystems using the framework. The framework itself does not impose theoretical constraint, and the use of the term *module* here should not be taken to imply a commitment to, for example, Fodorian (1983) modules. Rather, a framework module is the software component that permits the specification of the dynamic relationships among state variables. The extent to which a given set of state variable and function modules realizes an encapsulated module or a fully interactive subsystem depends on the details of the connectivity between state variables. (And as we see below, the architecture-to-brain mapping need not even imply strict localization of function).

Simulating the dynamics of an architecture

All of the components are run asynchronously with the architecture developer having complete control over when components start executing and how long they execute. The developer can specify independently for each module:

- When a module initiates execution. Examples include: periodically (such as every 50 msec), whenever inputs change, or even some delay after inputs change.
- The length of simulated time it takes for a module to execute and for data to travel between modules. This can be a fixed number, such as 10 msec or can be dependent on input parameters, such as (1 msec * number of changed inputs).

The Storm framework automatically coordinates the execution of the components (function modules and state variables), following the temporal constraints declared for each of the components, freeing the developer from writing code that schedules the execution of the modules. Thus, when an architecture runs, the framework automatically schedules all of the components, initiates their execution and provides a complete trace of the temporal activity of every module and state variable, including behavior in an external task environment. All of the scheduling is based on simulated time, which depending on the calculations performed in the modules could be much slower, or possibly even faster than real time.

This layer of abstraction thus allows the modeler to focus on the control structure of the *brain architecture* rather than the control structure of the *simulation*. Importantly, the function modules may be flexibly implemented via arbitrary code in the underlying target language, but the modules do not interact by calling each other directly, and the modeler need not worry about how to manage their parallel execution. (This abstraction away from simulation control structure is a common property of simulation environments long used in other areas of science.)

Experimenting with an architecture

Storm's design makes it easy to quickly add or replace modules because all of the information about a module is local to that module. (A critical aspect of this locality is the distributed nature of the simulation control, above). This makes it possible for research groups to share modules as well as to have multiple implementations of a given module. For some experiments, it might be desirable to have a coarse, but efficient implementation of a module, or replace a small network of modules and variables with a single module that is extremely fast, but only approximates a given computation. Moreover, during early development a coarse model might be all that is available. In others cases, a very accurate, but slow implementation of a module can be used when detailed behavior is critical.

The individual function modules and state variables are created by the architecture developer using a standard programming language. The framework currently supports C++ and Java, but will soon support MATLAB and R. This makes integrating existing code simpler and allows a module developer to select a language that is particularly well suited to the behavior they wish to model. The framework is agnostic as to which language is used to specify modules and one could imagine supporting the use of neural modeling systems.

In order to provide maximum flexibility and efficiency, the framework is designed to run on multiple operating systems (Windows, OSX, Linux) and has underlying support for parallel execution, which supports multi-core computers and will support clusters. This is transparent to users, determined at runtime based on the available resources. The framework itself is lightweight and requires minimal computational resources.

Mapping onto the Brain

In order to compare the processing in the architecture with what is known about the brain, the framework must support the explicit representation of processing and communication in the brain. In Storm, the *brain mapping graph* formally declares assumptions about the physical substrates of the state variable, and by implication, the function modules. State variables are mapped to different physical structures and regions within the brain, which are then mapped to physical coordinates in a normalized brain coordinate system. In the current design only state variables are explicitly mapped to brain regions. Function modules are thereby implicitly mapped to regions based on the state variables they use (Figure 2).

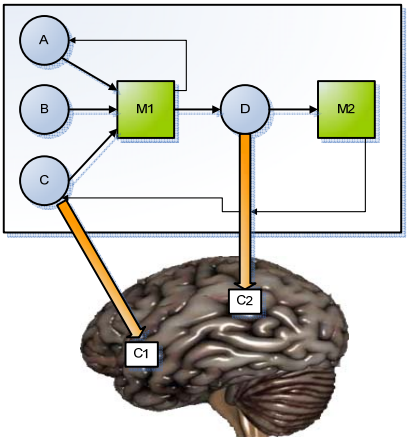

Figure 2: Brain Mapping Graph

This mapping scheme is quite flexible because there are no restrictions on what state variables might represent, and there are no restrictions on the target vocabulary of brain structures. For example, state variables might represent synaptic weights that could be changing over both the short and long-term, and such state variables might correspond to long-distance synapses in the brain that connect distal cortical areas. Or, a state variable might represent a quantity of some neurotransmitter that is fairly localized in space, or a vector of activation values representing patterns of firing activity in a particular part of the hippocampal formation, or an abstract short-term control symbol thought to be distributed over a broad area of prefrontal cortex.

Thus, this mapping scheme does *not* enforce a simple one-to-one mapping of computational function onto local structure. Rather, the mapping explicitly identifies the physical substrates of the state variables, and these physical substrates may be at any level of spatial resolution. The mapping of function to structure is then implicit in the mappings that function modules inherit from their state variables.

The functional connectivity graph together with the brain mapping graph imply a brain connectivity graph. That is, the connectivity of the state variables and function modules and their mapping to brain regions implicitly make claims about how the brain regions are connected, which can be tested against known constraints on how brain regions are actually connected. The predicted brain connectivity is derived from the connectivity of the function modules and the mappings from the architectural components to the brain as shown in Figure 3. These structures, together with the simulation provide Storm three important capabilities:

(1) Detecting inconsistencies between known brain connectivity constraints and the architecture.
(2) Predicting the time-course of activity in brain regions. This could support the automatic generation of simulated fMRI, MEG, or EEG for the modeled brain regions.
(3) Changes made to the architecture for functional reasons automatically change the biological predictions as the brain connectivity is derived directly from the functional elements of the architecture.

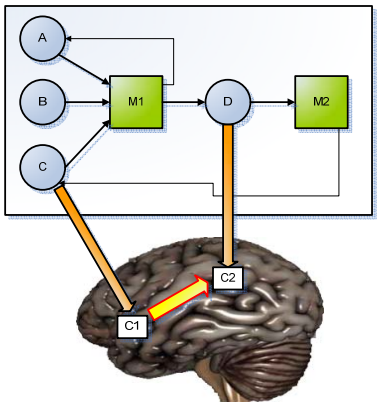

Figure 3: Brain Connectivity Graph

An Empirical Example in Using Storm

Example Task Requiring Learning and Memory

In order to demonstrate the Storm framework, we created a simple task that requires learning control knowledge for both internal and external actions. The example task is set in a 5x5 grid-world, shown in Figure 4. The domain contains three special locations, or *boxes*, in fixed positions: boxes A and B are *reward* boxes, while box M is an *information* box. The agent is rewarded with a positive reward when it opens one of the boxes and a negative reward when opening the other. The agent perceives a symbol when it opens the information box; this symbol is correlated with the location of the positive reward box (but does not correlate to any perceived feature of the boxes). An agent that cannot maintain the symbol in an internal memory would be unable to receive the maximum reward in every episode, making the task un-learnable.

The agent can move in the four cardinal directions, and if a box is in its current location, the agent can open the box. The agent perceives its location in the grid and any reward signal, but cannot perceive the labels on the boxes (A or B). If the agent is in the information box square and the box is open, the agent also perceives a symbol. An episode concludes when the agent opens the box containing the positive reward. The location of the rewards is randomized between episodes.

Reward is structured such that a positive reward has magnitude of +10, a negative reward is -10, and on every step that the agent does not open a reward box, it receives -1 reward.

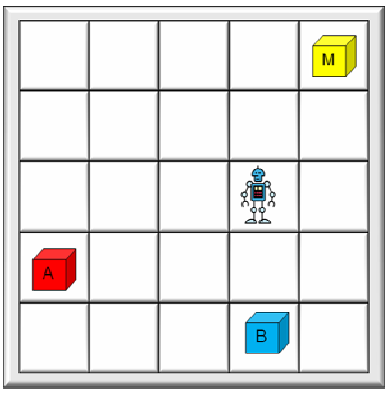

Figure 4: Information Box Task

An Example Architecture Developed using Storm

In order to help illuminate some of the framework's capabilities, we used Storm to develop a simple architecture capable of supporting an agent that learns to perform in the example task. This architecture combines a simple long-term memory with a basic reinforcement learning mechanism that learns control knowledge for both internal and external actions.

Our example architecture is shown in Figure 5. Function modules in the figure are represented as rectangles, and state variables as ovals. In this model, the environment is represented as a function module (for convenience) which receives a motor action as input and generates sensory information as output. Sensory Input is used by both Long and Short Term memories, which in turn is used by Action Selection to choose an internal Long Term Memory retrieval as well as an external Motor action. The Reinforcement Learning mechanism uses Working Memory, the Internal Reward Signal, and selected actions to adjust the control knowledge used by Action Selection.

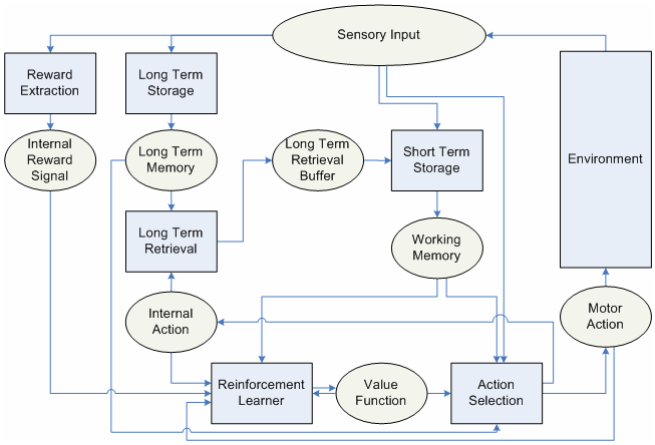

Figure 5: Simple Architecture Implemented in Storm

The details of each function module are described below. The Reward Extraction module reads the explicit reward generated by the environment in Sensory Input and stores it as an Internal Reward Signal. Long Term Storage stores any perceived symbol (i.e. the contents of the information box) to Long Term Memory. Long Term Retrieval retrieves a symbol corresponding to the Internal Action from memory and stores it in the buffer. Short Term Storage reads the agent's location from Sensory Input and the contents of the Long Term Retrieval Buffer and puts the concatenation of both in Working Memory.

The agent decides how to act in the Action Selection module, which uses Working Memory and the Value Function to select its actions (using a decaying epsilon-greedy strategy). The Value Function is a table that associates a pair of internal and motor actions with the contents of working memory and the estimated future reward of applying those actions. The Value Function is adjusted by the Reinforcement Learner with Sarsa (Sutton, 1996) based on input from Working Memory, Internal Reward Signal, and Internal and Motor Actions.

In the example architecture, function modules initiate their processing when their input state variables change, and all take a fixed amount of time to process and create results. During execution, many of the modules will execute in parallel, such as those that depend on Sensory Input. Others execute in sequence because of the dependencies of their input variables on other function modules. This parallelism enables an agent to perform internal and motor actions simultaneously.

An execution trace of the example architecture's function modules is shown in Figure 6, which is generated from the execution logs by a Storm utility. There are four different types of events logged by Storm for a function module.

- *RequestWakeup*: can occur on the time step when an input variable's value changes,
- *Wakeup*: occurs on the time step immediately following a *RequestWakeup* event (unless explicitly delayed),
- *Finished*: occurs on the time step on which the module sets its output variables and completes processing, and
- *Processing*: this is the time that a module is inferred to be processing between *Wakeup* and *Finished* events.

Multiple events that occur on the same time step are plotted as one event (e.g. all Environment events occur on the same step).

Figure 6: Sample execution trace of the example architecture as generated by a Storm utility.

Processing begins in the Environment module which sets the Sensory Input state variable (see Figure 5). When Sensory Input is set, a *RequestWakeup* event triggers the Reward Extraction, Long Term Storage, and Short Term Storage modules. All three modules then process in parallel, after which they set their respective output variables.

The Reinforcement Learner module next begins processing, as it relies on the Internal Reward Signal set by the Reward Extraction module. Similarly, the Action Selection module

relies on the output of the Reinforcement Learner, and the Long Term Retrieval module on Action Selection, which explains the serial behavior seen in Figure 6. This behavior arises from the dependences of the input variables of each module, and is not explicitly timed or engineered. However, the Environment module *is* configured to process periodically, which explains why it does not begin executing at the same time as the Long Term Retrieval module even though inputs for both modules are set by Action Selection.

Although the mapping of state variables to brain regions is an important commitment made in Storm, this example architecture is so simple that we do not hypothesize a mapping. Rather, the purpose of this example is to illustrate the framework's specification and simulation capabilities.

Results

Example Architecture We developed two agents in the architecture to perform the example task, one that automatically retrieves the information symbol from long-term memory (when available) and one that must learn to retrieve it. The performance of the two agents is shown in Figure 7. Asymptotically, the behavior of both agents is the same: the agent moves directly to the information box, opens it, and then simultaneously retrieves the identifying symbol from long-term memory while navigating to the positive reward box, opening it upon reaching its location. The results indicate that learning both control knowledge for an internal action in addition to an external motor action is not significantly more difficult than for an external action alone.

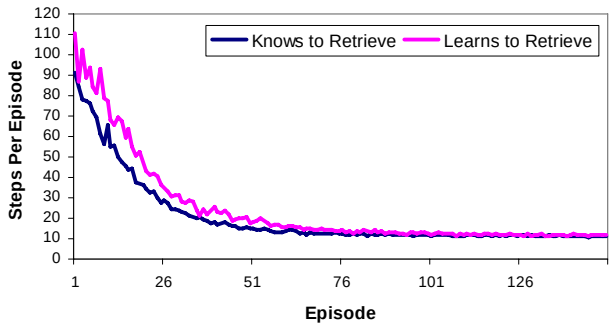

Figure 7: Learning curves for two agents performing on the simple task, average of 225 trials.

Modified Task and Expanded Working Memory To study the flexibility of an architecture using the Storm framework, we modified the task so that the agent had to learn to manage long-term memory retrievals:

- Instead of a single motor action to open a box, the agent now has two available actions. When the correct one is used to open the positive reward box, the standard reward is still received. However if the reward box is opened with the other action, a smaller positive reward (+1) is received.
- The information box contains an additional symbol identifying the correct action to use when opening the positive reward box. Both symbols are still automatically stored to long-term memory.

After an agent using the example architecture opens the information box, both symbols are then automatically stored to long-term memory. However, the Long Term Retrieval Buffer (and thus Working Memory) can still only store one retrieved symbol at a time. The agent therefore must learn to recall the two symbols at different times: the symbol identifying the correct box during navigation and the action symbol on the step before it will open the box.

We tested an agent using the example architecture as well as an agent with an expanded working memory that can store the two most recently retrieved symbols in working memory on the modified task. The results for both agents are presented in Figure 8. Although the agent using the architecture modified with an expanded working memory learns more quickly than the agent using the unmodified architecture, these results show that an agent using the unmodified architecture with limited working memory is still able to learn the modified task.

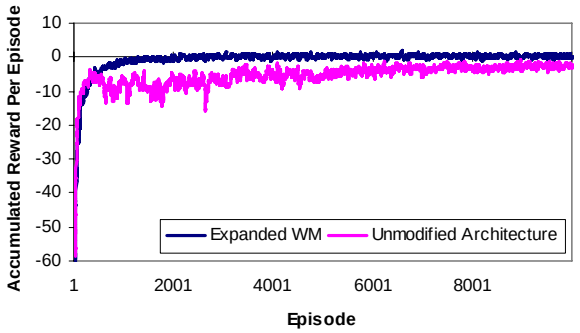

Figure 8: Learning curves for agents performing on the modified task, 25 per. moving avg. of medians for 45 trials.

In order to modify the architecture with an expanded working memory, only the Short Term Storage function module and Working Memory state variable needed to be changed – the rest of the architecture's function modules and state variables remained the same. This demonstrates an advantage to experimenting within the Storm framework: the modularized approach to development leads to architectures that can be modified quickly and easily.

Architectural Delay In our example architecture, all function modules took the same constant amount of time to process data (5 units of time as seen in Figure 6). In order to experiment with function modules processing at different time scales, we introduced a delay to the Long Term Retrieval module: with a delay, the module processes for 20 units of time rather than 5. This change has two effects: first, retrieved memories are available two environment steps after the Internal Action is selected; second, retrieved symbols in the buffer persist for two environment steps. Because of these changes, the agent can improve its performance by learning to make a retrieval from Long Term Memory two steps before it gets to the reward box.

The results of two agents, one modified with a delayed retrieval and the other unmodified, in the modified task are shown in Figure 9. While the agent using the unmodified architecture initially learns more quickly, the behaviors are indistinguishable after the 2000th episode.

In order to experiment with delaying Long Term Retrieval in the architecture, our implementation in Storm required only a single line of code to be changed. Storm's mechanism for scheduling the processing of function modules makes changing timing constraints to be a straightforward exercise.

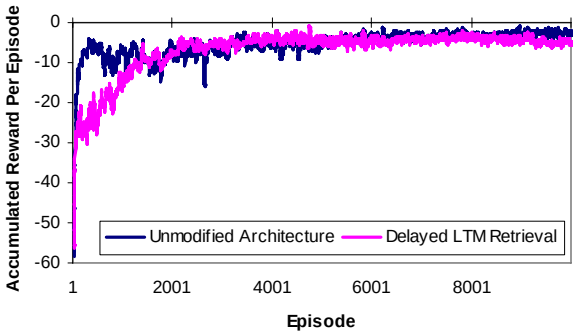

Figure 9: Learning curves for an architecture modified with delayed Long Term Retrieval compared with an unmodified architecture, 25 per. moving avg. of medians for 45 trials.

Summary of Experiments The Storm framework has allowed us to experiment with the example architecture in several dimensions, the results of which are not all shown in this paper:

(1) We experimented with two timing conventions: both waking function modules when input variables have been set as well as function modules processing periodically at set intervals. The example architecture implements a hybrid approach and uses both approaches in its modules.

(2) We experimented with the timing of individual modules, delaying their output such that the processing time of various modules overlaps.

(3) We explored reinforcement learning modules implementing a variety of learning algorithms with various parameter settings; switching algorithms is as simple as changing the module used by the framework.

(4) We have simulated environments in C++ function modules and interfaced to external Java environments.

When experimenting along all of these dimensions, the necessary changes to function modules were minor and no changes to the framework were necessary. In contrast, experimenting with existing cognitive architectures to modify the behavior of working memory, long-term memory, or timing constraints can often be difficult and time consuming.

Discussion

By developing our example architecture using the Storm framework, we have had valuable experiences which begin to shed light on the advantages (and disadvantages) of using a lightweight framework to model brain function.

The Storm framework has minimal overhead so as to not impede the development of a diverse set of functional modules. The framework does, however, strictly enforce that any data shared between function modules must be contained within state variables: designers must be explicit and consistent in the organization of data into state variables.

Modeling the timing of function modules and state variables is an important aspect of the framework and is straightforward to use and experiment with. This allows a designer to focus on implementing behaviors and not be concerned with the implementation of timing constraints.

One possible disadvantage of using the framework is the strict enforcement on the organization of data into state variables. Experimental architectures may not want to make strong commitments to the separation of data; algorithms achieving high-performance may also require a high level of abstraction as realized in function modules and state variables.

In the future we plan to begin testing Storm's ability to scale by building iteratively larger and more complex architectures, as well as developing psychologically plausible models using state variables that map to brain regions and model brain function.

Acknowledgments

The authors acknowledge the funding support of the DARPA "Biologically Inspired Cognitive Architecture" program under the Air Force Research Laboratory "Extending the Soar Cognitive Architecture" project award number FA8650-05-C-7253.

References

Alexander, A., Arbid, M. & Weitzenfeld, A. (1999). Web Simulation of Brain Models. *Proc. of the 1999 International Conference on Web-Based Modeling and Simulation*, 29-33. The Society for Computer Simulation International, San Diego, CA.

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

Bothell, D. (2004). ACT-R 6.0 implementation. http://act-r.psy.cmu.edu/actr6/

Eliasmith, C. Anderson, C. H. (2002). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. MIT Press.

Fodor, J. A. (1983). *Modularity of Mind: An Essay on Faculty Psychology*. Cambridge, Mass.: MIT Press

Gray, W. (2007). *Integrated Models of Cognitive Systems*, Oxford University Press.

Kieras, D. & 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.

Munakata, Y., & Johnson, M. H. (Eds.) (2006). *Processes of Change in Brain and Cognitive Development: Attention and Performance XXI.*, Oxford: Oxford University Press.

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

Simen, P., Polk, T. A., Lewis, R. L. & Freedman, E. (2004). A computational account of latency impairments in problem solving by Parkinson's patients. *Proceedings of ICCM 2004*, Pittsburgh.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (Eds), *Advances in Neural Information Processing Systems: Proc. of the 1995 Conference*, 1038-1044. MIT Press.

Weitzenfeld, A., Arbib, M., and Alexander, A. (2002), *The Neural Simulation Language: A System for Brain Modeling*. MIT Press.