Deconstructing ACT-R

Terrence C. Stewart (terry@ccmlab.ca)
Robert L. West (robert_west@carleton.ca)
Carleton Cognitive Modelling Lab
Institute of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6 Canada

Abstract

To broaden the use and understanding of ACT-R within the cognitive science community, we have attempted to express the system in terms of a large number of simple, interacting components. We re-implemented each of these components from the set of formulas that make up ACT-R theory, but used an alternate syntax for the expression and simulation of cognitive models. Care was taken to ensure the new syntax is as explicit as possible in terms of the claims being made and the processes occurring within the model. Generally, people view ACT-R as a production system; however, in deconstructing it we found that it is significantly more than this.

Introduction

Cognitive modeling, as a science, involves not only building and evaluating models, but also effectively communicating the results and the implications of those results. Communication becomes increasingly more difficult as the audience knows less about the particular approach that was used. It is easier for neural network people to communicate to other neural network people than it is for them to communicate to production system people, and it is even harder to communicate to people who do not use cognitive modeling at all. In particular, we have found that non-modelers have a very poor understanding of the concept of *cognitive architectures*.

The concept of a cognitive architecture was put forward by Newell (1990) to deal with the problem that the behavioral sciences, and psychology in particular, study the mind by dividing it up into specialized sub-fields, without attempting to assemble the results into an integrated model of the mind. Cognitive architectures are meant to be a way to do this. Anderson (1993) further clarified the concept of cognitive architectures with his distinction between frameworks, theories, and models; where frameworks are general claims about cognition, theories are specific formulations about how the frameworks operate, and models are the theories applied to specific tasks and behaviors. Therefore, cognitive architectures are theories about how the mind integrates different processes to produce thoughts and behaviors.

Because cognitive architectures tend to be complex, they are often expressed as computer programs. Within the cognitive modeling community this is generally regarded as a good thing because it means that the theory is precisely specified, avoiding ambiguity and vague statements. However, outside the community it is often viewed with suspicion. Indeed, it is often felt that modelers merely write computer code that mimics the human data (i.e., that modeling is merely descriptive). In order for non-modelers to be convinced that this is not happening, we need clear descriptions of the architectures and how they are used.

Furthermore, it needs to be clear that while cognitive architectures provide a language for writing the models, the architectures also constrain model building so that the resulting models are products of the theory behind the architecture. However, it is generally the case that the only people with enough knowledge to check this are the people who developed the system in the first place. Also, the initial implementation of the architecture tends to be tied to the language and programming style of these researchers.

Python ACT-R

We believe that re-implementing cognitive architectures is an effective way to demonstrate and clarify exactly what they do. This effect is in addition to the advantages for error-finding (as mentioned in Erev & Barron, 2005), and is related to model aligning (Axtell et al, 1996). To illustrate this we give an overview of Python ACT-R, which is our re-implementation of the standard Lisp ACT-R, using the Python language.

The Python ACT-R project was initially undertaken to gain a more complete understanding of the ACT-R theory. Creating such a system also brings to light aspects of ACT-R which may not have received much attention, and assures us that there are no hidden aspects influencing results. In doing this, we made Python ACT-R different from Lisp ACT-R in three major ways.

First, we attempted to break ACT-R into individual components, each small enough to be described by a simple algorithm. These components were implemented separately and then combined into a full ACT-R system, making each component distinct and easily accessible. Interestingly, this means a modeler can assemble different variants of the ACT-R that they wish to use while building their models. This is in contrast to the Lisp ACT-R process of turning features on or off.

The second major difference was that we modified the syntax for writing ACT-R models. This was originally done to ease the integration of ACT-R into the Python language. However, as a byproduct, it also made the distinction between the core ACT-R theory and the particular syntax used to write models more clear. The result is a system that is functionally equivalent to Lisp ACT-R, but does not make the same implementation and syntax choices.

Our third difference was to be as explicit as possible in terms of what is happening within the model. In Lisp ACT-R, there are many side-effects: situations where code in the model that explicitly does one thing also causes one or more other actions to be performed that are not explicitly represented in the model code. This leads to programming efficiency and makes certain parts of the theory automatic,

but it can also be confusing to less experienced modelers, especially because these side-effects are subject to change between versions of ACT-R. Instead, we decided to be more explicit, and possibly less efficient, by eliminating side effects and requiring the modeler to specify what actions are happening at each stage.

Also, we have not re-implemented the complete ACT-R perceptual-motor system (ACT-R/PM) and we have not fully re-implemented production compilation, so we will not be discussing these in detail. Since we have not examined these aspects, it is possible that they contain some exceptions to our interpretation of the ACT-R theory, which is based primarily on the overall computational architecture, the production system and the declarative memory system. Also, although our picture of the ACT-R theory is based on the Lisp code, it is still our interpretation, so we make no claims that it fits with official ACT-R orthodoxy.

The description is divided into three parts. The first part describes the system that allows the modules of ACT-R to communicate, the second part describes the modules, and the third part describes additional functions that are required for it all to run.

Communication between Modules

ACT-R is a modular theory of mind. That is, it treats the mind as being composed of distinct modules that exist for particular functions. This being the case, the modules need to communicate to each other using a common mechanism. We chose to describe this communication system first, rather than following the approach taken by other papers on ACT-R, which tend to start with the production system module, because we believe this aspect of ACT-R is under appreciated.

Chunks

In ACT-R the various components of the architecture communicate using a simple symbolic representation system, called a *chunk*. Each chunk has a number of *slots*, each of which contains a single symbol. These symbols can represent anything (including other chunks), but do not have inherent semantic value. As a guideline, it is recommended that chunks have a small number of slots. Miller's number of 7±2 is recommended as an upper limit (Anderson & Lebiere, 1998), although this is not enforced.

To do anything with the slots of chunks, there must be a way to distinguish between them. In Lisp ACT-R this is done by giving each slot a name. Python ACT-R allows for this, but the default is for slots to be distinguished by position instead. For example, the following text shows how a chunk representing a large, friendly dog might be represented in Lisp ACT-R and Python ACT-R. Note that we will use the Python ACT-R syntax throughout this paper.

Lisp: `(chunk isa dog size large manner friendly)`
Python: `chunk='dog large friendly'`

In Lisp ACT-R, one particular slot is treated differently from the other slots. The `isa` slot (the first one in the above example) is meant to indicate what *type* of chunk it is. Chunk matching (as described later) to the `isa` slot used to be necessary for Lisp ACT-R to work, but this is no longer required in ACT-R 6. This change suggests that the special treatment of the `isa` slot is no longer a core component of the ACT-R theory, so we have not included it in Python ACT-R as a computationally enforced modeling constraint.

Buffers

The other component used for communication between modules is the buffer system. Buffers are capable of holding only one chunk at a time. Modules can place a chunk into a buffer, modify the value of slots of a chunk, or clear the buffer. They can also retrieve a copy of a chunk in a buffer. All of these actions are performed instantly. Importantly, the chunk it contains is a *copy* of the original chunk, meaning that changes to the contents of the buffer do not alter original chunk. Taken together, the buffers create a representation of what might be called context. That is, because they hold the most recent output of each module, they more or less collectively represent the current state of the whole system. This represents a strong claim of the ACT-R theory, that the state of the system is represented by a limited number of chunks in the buffers. Also, the buffers are considered to be physically separate from other ACT-R modules (Anderson et al, 2003). This means that the buffers represent particular areas in the brain.

Chunk Matching

One of the fundamental mechanisms for working with chunks is to be able to find *matches*. There are two ways this is used; first, to examine the current context (as defined by the contents of the buffers) to determine what action to take next, and second to search for a particular chunk among a large collection of chunks (such as might be stored in declarative memory). The details of such usage are described later, in the sections about the appropriate modules. However, the fundamental mechanism is common across these situations.

In all cases, the matching process involves determining whether or not a particular chunk matches to a particular pattern. A *pattern* is a chunk whose slots contain matching rules, rather than actual contents. A pattern matches with a given chunk if each of the slots in the pattern has a rule that matches with the contents of the corresponding slot in the chunk.

There are three matching rules[1]. The first is an exact match, where the pattern slot indicates the exact content that must be in the chunk slot. Second is a 'not' match, where the pattern indicates a value that the slot cannot have. In the python syntax we use a `!` to indicate the 'not' match, so the pattern `'dog !small friendly'` would match with the chunk `'dog large friendly'` but not `'dog small friendly'` or `'dog large vicious'`. The third matching rule matches to the situation of having no chunk at all. This handles the special case of matching to an empty buffer.

Also important for the matching process is the idea of a *variable*. This is an arbitrary label that can take on different values when it is placed in the slot of a pattern. The first

[1]There are modules that have other sorts of matches, such as numerical comparisons. These module-specific additions are not necessary for implementing the core theory of ACT-R.

time a variable is used within a match, its value is *bound* to the value in the corresponding slot. In Python ACT-R, we use a `?` to indicate variables. For example, given the chunk `'dog large friendly'` and the pattern `'dog ?size friendly'`, the variable `?size` would be bound to the value `large`. Once a variable is bound, further uses of that variable within that context must have the same value. Thus, the pattern `'cat ?size sleepy'` would now match to the chunk `'cat large sleepy'` but not `'cat small sleepy'` because the variable `?size` has been bound to `large`. One can also use an `!` to indicate a not-match for variables (such as `'cat !?size sleepy'`).

Matching patterns are directed at specific buffers. Every chunk/pattern pair must match for there to be an overall match, and the variable binding system works across all patterns. For example, if the matching pattern was:

```
buf1='dog ?size friendly',buf2='cat ?size sleepy'
```

it would match to either of the following buffer contents:

```
buf1='dog large friendly',buf2='cat large sleepy'
buf2='dog small friendly',buf2='cat small sleepy'
```

but would not match in this case:

```
buf3='dog large friendly',buf2='cat small sleepy'
```

The variables and matching rules can be combined in any manner, including having multiple rules for the same slot (in which case all rules must match; the main use for this would be multiple 'not' matches, e.g., to match to an animal that is not a dog and not a cat).

Partial Matching

While the above description handles the majority of the pattern matching ability in ACT-R, there is one further optional complication. There is a mechanism called *partial matching* that must be considered when implementing the chunk matching system. With partial matching it is possible to indicate that certain chunk slots do not have to match their corresponding slot in the pattern for a match to occur. This mechanism is only currently used by the ACT-R declarative memory but it requires such tight integration with the general chunk matching system that its functionality is best defined there.

With partial matching, the matching system will not just indicate whether or not a particular chunk matches the given pattern, but it will also indicate a numerical *matching penalty*. To configure this, each slot gets a value *P* that indicates how important it is to match. This is either a numerical value, or a special value that ensures that partial matching does not apply to this slot. This *P* value is set based on the name of the slot, can be set by the modeler. The calculation of the matching penalty is then a simple sum.

$$\sum_k P_k M_k$$

Here, k represents each slot of a chunk in turn. M is an indication as to whether or not the value in the chunk matches what was expected by the pattern's matching rule. This value is 0 if it does match, and defaults to 1 if it does not match. This value can also be set for particular combinations of values. For example, a modeller might indicate that a chunk value of 'pink' might only give a 0.5 penalty when the pattern is trying to match on 'red'. The resulting penalty value can then be made use of by whatever module is performing a match. In the case of declarative memory, this value is subtracted from the activation level (described later) of the chunk.

Modules

Modules in ACT-R represent modules in the brain. Therefore, like the buffers, there is an expectation that they correspond to brain areas. ACT-R theory, therefore, divides the brain in two types of systems: functional systems (modules) and communication systems (buffers). The small chunk size recommended for the buffers can be seen as representing limited channels of communication between the modules. Modules are implemented such that when conditions require the use of a module it is activated instantaneously, in simulated time (obviously this takes some amount of computational time but it is not added to the simulated time). Once activated, modules calculate the total cost in time for the action they are going to do. Once this amount of time has passed from when they were activated the action is instantaneously (in simulated time) carried out. All modules and buffers operate in parallel.

Production System

The ACT-R production system contains a collection of *if-then* rules for accomplishing tasks and coordinating cognition, perception and motor actions. The production system's job is to determine what production to *fire* (i.e. what action to carry out) at any given moment. Firing takes 50 milliseconds of time, and only one production can fire at a time. If no productions can fire, then the system does nothing. The productions are initially hand-designed by the modeler based on their theory as to how a particular task is performed, but the system can generate new productions through the production compilation mechanism (discussed later).

The *if* part of a production (referred to as the Left-Hand Side in the ACT-R literature) is a collection of matching patterns, as described above. Whenever these patterns match, the production can fire. Productions can be set to match on all the buffers or only a subset. Overall, this forms a constraint that any information being used for decision-making by the production system must be present in one of the buffers in the system.

In addition to this, the *if* conditions can also include matches to the state of a module. To accomplish this, a module can include a separate buffer that holds information about its state. The particular slots and contents for this state buffer are specific to each module, but would include information such as whether the module is currently performing an action.

The *then* part of the rule (the Right-Hand Side) consists of a series of actions to be taken when the rule fires. The actions are commands for the other modules or buffers. In the case of buffers, commands can include setting or changing the values of chunks within the buffers (or emptying the buffer). For modules, the commands (referred

to as *requests*) can trigger modules to perform any action that a module can do. ACT-R theory implies that there are restrictions as to how many of different sorts of actions may be performed on the right-hand side (e.g. one cannot make more than one request to declarative memory at a time).

Any bound variable from the *if* part can be used in the *then* part of the production, but after the production is fired the bound variable is discarded. This means that the power of using variables is limited in time and in space (i.e., variables created within the productions system module cannot be used outside of it).

In the implementation of this process, the production system *instantaneously* finds all the matching productions. This may require actual computation time, but it does not affect the elapsed simulation time (i.e. no other actions are happening while the search for matching productions occurs). If a match is found, then the production system *waits* for 50 milliseconds. During this 50 milliseconds, the other modules may perform various actions (such as retrieving chunks from declarative memory, or moving attention in the visual system). After this, all the commands on the *then* side of the production are executed. This is also treated as an instantaneous action. However, the requests generally will not be immediately carried out as each module calculates the time delay for the actions. Once the requests are made, the production system searches again for new productions to fire.

If no productions match, then the production system does nothing. However, the moment a buffer's content *changes* such that a production could fire, the production system notices this and readies that production to fire 50 milliseconds later. The production system must thus perform its search for matching patterns any time there is a change to the buffers and it is not currently waiting to fire a previously matching production.

As far as we can see, the production system must also guarantee that the contents of the buffer *are still a valid match* after the 50 millisecond delay. That is, even if the contents of the buffers have changed since the beginning of the 50 millisecond delay, the changes must not be ones that affect the matching, otherwise the production could be inappropriate and could also crash the system. In Lisp ACT-R it is unclear to us whether the pattern must match *at all times* within the 50 millisecond time frame, or just at the beginning and end. Also, it is unclear if the production system restarts *immediately* upon a buffer change, or if it waits until the 50 milliseconds are over. If this occurs in Python ACT-R the production selection process is restarted without firing the previously selected production.

Another issue is that in Lisp ACT-R requests are sent to modules by placing the requests in a buffer associated with the module, whereas in Python ACT-R the requests are sent directly to the module. Using the buffer in Lisp ACT-R does not impose any time delay, so the only constraint it imposes is that the request should be a chunk, and thus of limited size. This constraint is no different for Python ACT-R, and is not enforced in either of them. There is thus an asymmetry in the use of buffers; they are needed for storing data from the modules, as this data needs to persist across time. However, messages telling the modules to perform actions do not need to persist, and so do not need a storage location.

Production Conflict Resolution

The one issue we have not addressed thus far is what to do when multiple productions match. Since only one can fire at a time, there needs to be a method for choosing which one. In ACT-R, this is done by estimating a production's *utility*. Whichever matching production has the highest utility is the one that will fire. The standard approach for calculating utility is based on the following formula:

$$U_i = P_i G - C_i + \varepsilon$$

The three parameters are: the probability that the production will lead to a *success* (*P*), the value of that outcome (*G*), and the amount of time that will occur between performing this production and achieving that outcome (*C*). Both *P* and *C* are specific to a given production, and *G* is an overall parameter for all productions. ε is an optional random noise value with adjustable variance, usually set to 0.

It is possible to simply set each of these values manually, but it is desirable to create models that will *learn* values for these parameters on their own. If this mechanism is used, then it is necessary to indicate *successes* and *failures*. In ACT-R, this is done by marking certain productions as indicating success, and others as indicating failure. The system keeps track of the number of times that firing each production eventually leads to a success (*s*), and how many times it leads to a failure (*f*). It also keeps track of the total amount of time between a production firing and either a success or a failure occurring (*t*). Given this data, *P* and *C* are estimated as follows:

$$P = \frac{s}{s+f} \qquad C = \frac{t}{s+f}$$

There are also alternate versions of these formulae, which include the ability to record multiple successes and failures at once (Gray, Schoelles, and Sims, 2005). Since this is a developing area of research, Python ACT-R includes these variations, and makes it easy to create and modify mechanisms for adjusting the utility of productions.

As a final note, ACT-R also defines a system for the generation of new productions out of old ones. Here, two productions are combined into a single production that performs both actions. Importantly, this is only done in situations where *the two productions would have fired one after the other.* This mechanism is complex enough to take into account situations where the second production fires as a result of a declarative memory retrieval (as will be discussed in the next section), and so is meant to model the transition from having to explicitly recall what to do next to simply automatically performing that action. A Python ACT-R version of this mechanism is still being developed.

Declarative Memory

The declarative memory system in ACT-R is a module for storing and retrieving chunks. There is no limit on its capacity, other than its mechanism for making chunks harder to retrieve the less often they are used.

Adding a chunk immediately places it into whatever internal storage mechanism is being used in the given implementation. If two identical chunks (i.e. ones with all slots with identical values) are added, then the system treats them as the same chunk. This mechanism is called *merging* and is used to strengthen the activation of the chunks, as described later.

In Lisp ACT-R, chunks are normally added at the very beginning of the model (representing background knowledge). There are then various rules indicating when the chunks in various buffers should be *automatically* added to declarative memory. As discussed above, in Python ACT-R the modeler needs to explicitly state when chunks should be added. That is, in Python ACT-R we rely on the modeler rather than the software to implement this aspect of ACT-R theory. This makes it easier to write "illegal" code, but it also makes it easier to experiment with this part of the theory.

When requesting a retrieval from memory, a pattern (as described above) is given to the module. The declarative memory system will then find the chunk that matches the pattern and has the highest *activation* value, and place it into a particular buffer known as the *retrieval* buffer. The amount of simulation time taken is based on the following formula

$$t = Fe^{-A}$$

Here, F is a parameter called the latency factor, which is set by the modeler. A is the *activation* of the chunk. If no chunks have an activation higher than the *threshold* (which is also set by the modeler), then no chunk is placed in the buffer.[2] The amount of time this takes is determined by substituting the threshold into the above equation instead of A.

There are a variety of mechanisms in ACT-R for determining the activation of the chunks. They can be set manually, but this practice is very rare. Almost always they have a certain amount of random noise associated with them. The main method for adjusting the activation is known as base-level learning. Here, the activation is adjusted based on the use of the chunk. When a chunk is used (i.e. whenever an identical chunk is added), this increases its activation. Conversely, the activation level gradually decreases when it is not used. Exactly how this process is best modeled is a subject of much study, resulting in a number of different approaches:

$$B = \ln\left(\sum t_i^{-d}\right)$$

$$B = \ln\left(\frac{n}{1-d}\right) - d\ln(L)$$

$$B = \ln\left(\sum t_i^{-d_i}\right), d_i = c\,e^{-A} + a$$

The first formula is based on extensive experimental results (summarized in Anderson & Lebiere, 1998). t_i represents the times in the past that this chunk has been added (measured in seconds before now). The parameter d controls how quickly the activation decays, and is always set to 0.5 to match known human data. The second formula is an approximation of the first, where n is the number of times in the past it has been used, and L is the amount of time since the chunk was first created. This formula tends to be used for reasons of computational efficiency, but recent results (Sims & Gray, 2004) have shown that it gives significantly different results from the first equation in certain situations and is therefore problematic.

The third formula is a more recent system created by Pavlik and Anderson (2005) that allows d to vary based on the current activation of the chunk. Using a chunk when it has a high activation leads to faster decay, while using it when it has low activation gives less decay. Instead of specifying d, this requires specifying two parameters, c and a. The resulting behaviour more closely matches spacing effect phenomena in memory studies (Pavlik & Anderson, 2005). All of these are available in Python ACT-R.

Additional Functionality

There remains one major aspect of ACT-R that has not been described, and which tends to receive relatively little attention. This aspect is the overall framework which allows researchers to define ACT-R models, controls how the components within those modules interact, and generates data which can then be compared to data from the real-life situation being modeled. Taken together, this can be seen as the *operating system* that the model is defined within.

The goal of this sort of background framework is to be as *theory-neutral* as possible. The particular decisions made in implementing it should have *no impact* on the predictions of the model. Indeed, if it is discovered that there are aspects which do affect the model, then we need to identify those aspects and analyze them to combine them with the explicitly described theory (for an example of this, see Axtell et al, 1996).

The first component of the framework is how models are defined. In Lisp ACT-R, this is done by defining a specialized language using the Lisp Macro system. In Python ACT-R, the built-in class definition system is used, with functions for productions and objects for the various components. Each allows for the customization of the model in different ways (Lisp by a large collection of parameters, and Python by declaring which modules are to be used). Each of these approaches has advantages and drawbacks, but they are importantly functionally identical.

The next consideration is the *scheduler*, which allows for the simulation of multiple parallel modules. Any parallel cognitive model (including ACT-R) requires a method for saying that a particular action will take place at a particular *time* in the future, and having that time be tracked internally, rather than using the actual time.[3]

[2] If this occurs, the module is said to be in an *error* state, which is one of the slots in its implicit state buffer (along with whether or not it is *busy*), as described earlier. This error slot will continue to be true until the next retrieval request is made.

[3] This is generally not a system that exists in most programming languages, although it can be implemented easily using modern techniques such as greenlets or continuations.

Two other important components are the logging system (which identifies what is happening within the model, allowing for analysis of the temporal order of internal events) and the mechanism for connecting an ACT-R model to an environment. These can get technically complicated, but. as long as they work, they do not have any impact on experimental results.

Implementation

All of the afore-mentioned ACT-R components have been implemented in the current version of Python ACT-R (Stewart & West, 2005) available at <http://ccmlab.ca/actr>. Also included are basic visual attention and motor modules, and an SOS-based mechanism (West and Emond, 2002) for easily creating specialized sensor and motor modules for custom environments.

As described in (Axtell et al, 1996), computational models can best be shown to be compatible with each other by showing that their resulting data is statistically indistinguishable. To validate Python ACT-R, we are thus collecting a variety of Lisp ACT-R models, creating equivalent Python ACT-R models, and comparing their results. We have started our comparisons with the models in the official ACT-R tutorials. We have found that Python ACT-R does not differ from Lisp ACT-R for any of the models in tutorial units 1, 2, 4, and 5 (unit 3 deals with details of the vision system, and tests of units 6 and 7 are underway).

Conclusions

Through this project, we have been forced to re-examine what ACT-R actually is. Deconstructing and re-implementing it has highlighted for us the various different uses of ACT-R. First, ACT-R is a comprehensive theory of human cognition, including how cognition interacts with other mental processes and with the environment. From this perspective, ACT-R specifies what modules exist, how they work, and how they interact with each other. This includes very specific claims expressed as mathematical formulas, such as the PG-C utility learning rule or the declarative memory base level activation formula. Importantly, this is a work in progress; as more evidence accumulates, formulas are modified and modules are added. Thus ACT-R is not a fixed theory about cognition.

In terms of understanding such an extensive theory, the modular structure is important as it allows the overall theory to be broken down and understood by examining the individual modules. In this sense, each module is itself a theory about how that aspect of cognition works. In re-implementing ACT-R we did not copy the Lisp code but rather inserted the ACT-R formulas into the Python ACT-R modules. The fact that this worked shows that the ACT-R modeling system is what the ACT-R theory claims it to be.

Another way to view ACT-R is as a tool for building computational models of cognitive behavior. That is, the structure of the code allows for the development of cognitive architectures based on the principles and limitations described in the first section on communication. Modules can be added or deleted, as can the routes of communication between the modules. This can provide a valuable test bed for psychological theories, which tend to be about modules. The ACT-R modular system provides a structure for embedding these theories within a larger theory of the mind to (1) see how that integration could work and (2) be able see the effects of the module in complex tasks that require the use of other modules. The other modules could be the ACT-R modules, modified versions of the ACT-R modules, or completely new modules. For example, there is no reason why a module could not be implemented as a neural network or any other learning system.

Acknowledgments

Funding for this project was provided via a grant from the Natural Sciences and Engineering Research Council of Canada.

References

Anderson, J. R. (1993). Rules of the Mind. Hillsdale, NJ: Erlbaum.

Anderson, J. R., Qin, Y., Sohn, M-H., Stenger, V. A. & Carter, C. S. (2003). An information-processing model of the BOLD response in symbol manipulation tasks. Psychonomic Bulletin & Review. 10, 241-261.

Axtell, R., Axelrod R., J.M. Epstein and M.D. Cohen (1996), "Aligning Simulation Models: A Case Study and Results", Computational and Mathematical Organization Theory 1(2), pp. 123-141.

Erev, I. and Barron, G. (2005). On adaptation, maximization and the value of a cognitive interpretation of the Law of Effect. Psychological Review.

Gray, W. D., Schoelles, M. J., & Sims, C. R. (2005). Adapting to the task environment: Explorations in expected value. Cognitive Systems Research, 6(1), 27-40.

Newell, A. (1990). Unified Theories of Cognition. Harvard University Press, Cambridge, Massachusetts.

Pavlik, P. I. & Anderson, J. R. (2005). Practice and forgetting effects on vocabulary memory: An activation-based model of the spacing effect. Cognitive Science, 29, 559-586.

Sims, C. R., & Gray, W. D. (2004). Episodic versus semantic memory: An exploration of models of memory decay in the serial attention paradigm. In M. C. Lovett, C. D. Schunn, C. Lebiere & P. Munro (Eds.), 6th International Conference on Cognitive Modeling–ICCM2004. Pittsburgh, PA.

Stewart, T.C. & West, R. L. (2005) Python ACT-R: A New Implementation and a New Syntax. 12th Annual ACT-R Workshop

West, R. L., & Emond, B. (2002). Can cognitive modeling improve rapid prototyping. Carleton University Cognitive Science Technical Report 2002-05.