In D. Reitter & F. E. Ritter (Eds.), Proceedings of the 14th International Conference on Cognitive Modeling (ICCM 2016). University Park, PA: Penn State.

Cognitive Code: An Embedded Approach to Cognitive Modeling

Dario D. Salvucci (salvucci@drexel.edu)
Department of Computer Science, Drexel University
3141 Chestnut St., Philadelphia, PA 19104, USA

Abstract

For several decades, production systems have been the dominant framework in which (primarily) symbolic cognitive models have been developed. This paper proposes a different approach, *cognitive code*, in which behavioral models are developed directly in a modern programming language. However, unlike standard code, cognitive code has simulated timing and error characteristics intended to mimic those of human cognitive, perceptual, and motor processes. Some of the benefits of this new approach are illustrated in sample models of a paired-associates task, reading task, and dual-choice task.

Keywords: Cognitive architectures; ACT-R

Introduction

Since their introduction decades ago, cognitive architectures (Anderson, 1983; Newell, 1990; see also Gray, 2008) have provided a rigorous computational framework in which scientists can build and run cognitive models. Most importantly, a cognitive architecture represents a "unified theory of cognition" (Newell, 1990) that allows detailed exploration of the integration among various human systems, including cognitive, perceptual, and motor systems. Cognitive architectures have facilitated major advancements in cognitive science for specific research domains such as list memory (Anderson, Bothell, Lebiere, & Matessa, 1998) and multitasking (Borst, Taatgen, & van Rijn, 2010); at the same time, architectures have been applied to real-world domains such as gaming (Laird, 2002) and driving (Salvucci, 2006).

Even considering these successes, adoption of cognitive models outside of the academic research community[1] has been limited. There are arguably several reasons for this limited adoption:

Programming Paradigm and Language. Production systems have been the dominant framework in which (primarily symbolic) cognitive models have been implemented. Production systems, based on representation of processes as condition-action production rules, have been key to certain theoretical claims (e.g., learning from instructions: Taatgen, Huss, Dickison, & Anderson, 2008). However, from the perspective of programmers outside the cognitive-architecture community, production systems are largely an unknown quantity; instead, modern programmers typically develop code using modern procedural and object-oriented programming languages (Java, C++, Python, etc.). Learning the very different programming paradigms and patterns used in production systems is a significant barrier to developing models, even for those with a significant computer-science or programming background.

Model-Centered Development. For those in the research community, the cognitive model is often the centerpiece of the main research effort, and a great deal of time and care is taken to develop these models. For this reason, modeling frameworks often include an integrated user-interface environment and suite of tools to facilitate model development. However, for an outsider looking to embed a cognitive model into their own project—say, a game-engine programmer who wants to develop a cognitive agent to embed into a larger game—the model is a peripheral component rather than a central one. For this audience, learning an entirely new modeling language and development environment can be a large investment, often too large to make it worthwhile.

Lack of Model Integration. Cognitive modelers, especially those interested in cognitive architectures, have long stressed the benefits of a community-driven approach to unified theories of cognition. Over the years, this integration has largely come about at the architectural level, with a wide set of models using a single architecture or framework to generate behavior. Unfortunately, integration among models themselves—for example, reuse of existing models to develop new models—has arisen much less frequently, partly because modeling frameworks have not emphasized rigorous formal APIs that are crucial for integration.

Modern cognitive architectures and models have much to offer beyond the boundaries of the research community; with a blend of psychological theory and computational simulation, they contain a breadth and depth of predictive accounts that can be widely useful for other research and practical domains. Yet, because of the reasons above, the investment needed to extracting predictions from these models is often too large for those with less than a primary interest in cognitive modeling.

[1] Companies such as Carnegie Learning and Soar Technology have successfully applied cognitive models beyond the walls of the research community; still, the impact of cognitive modeling pales in comparison to related research methodologies—for instance, machine learning—that have exploded in popularity.

Cognitive Code

Cognitive code is code embedded in a modern programming language that aims to simulate and mimic human cognitive, perceptual, and motor processes. Of course, cognitive models and architectures have long used computational representations to simulate human processes, but they have (as noted earlier) generally defined their own programming language for this purpose, and generally relied on production systems as a central tenet of their representations. In contrast, cognitive code is embedded and written directly in an existing programming language, using structures and patterns already familiar to most programmers. At the same time, cognitive code differs from standard code in that it includes new software design patterns and libraries to facilitate the simulation of timing and errors inherent in human processes.

Before delving into the details, let us illustrate the basic concept with a simple example. Consider sample cognitive code (here in Java) that stores a new piece of information into memory:

```
memory.store(new Chunk("cat")
     .set("owner", "Jane")
     .set("name", "Whiskers"));
```

Here we create a "chunk" of information (as in ACT-R: Anderson, 2007) that defines knowledge about Jane's cat, and we store it into memory. Later, we try to recall this information from memory with a query:

```
Chunk chunk = memory.recall(new Query("cat")
     .add("owner", "Jane"));
```

In each case, the code uses patterns that would be straightforward and recognizable by most programmers. At the same time, this code conceptually differs from standard code in two ways. First, each step incurs a simulated temporal cost that corresponds to the time needed to perform this action in the cognitive system. In our example, the `memory.store()` action incurs some time, say a few hundred milliseconds, to add this knowledge to memory, and the `memory.recall()` action also incurs an amount of time that may depend on many factors (time since learning, number of times practiced, etc.). Second, each step has some potential for failure that mimics a true cognitive system; for example, the `memory.recall()` action could fail and return a *null* result depending on the current state of the cognitive system.

We now describe a prototype system that embodies the cognitive code approach. The system is implemented in Java as a library that can be readily integrated into other projects.

Core Simulation System

The core system centers on the concept of an *agent* that acts within the simulated world. An agent consists of any number of *modules* that define behavior for a particular subsystem (e.g., memory, vision, etc.) or for a particular domain (arithmetic, driving, etc.). The basic unit of information shared across modules is an *item*, which comprises a set of slot-value pairs, one of which can be the "isa" type of the item. Modules can utilize *workers* to perform work for them, and can share information with other threads through *buffers*.

Because the passage of time is central to the cognitive code approach, we also require some way to simulate a central clock within the code. Within a single thread, we simply maintain a simulated time to be incremented by each cognitive step. In the general case, though, we need multiple threads to share a single clock; human multitasking will be best represented as separate cognitive threads, as discussed later, and even a single thread may require that certain operations happen in parallel (e.g., moving a hand while recalling information).

The core simulation enables the central clock as follows. The system assumes that individual operators on a particular thread can define their own delays, effectively stopping execution until the clock reaches the new time after delay. For example, let's say we run the following code on one agent thread:

```
agent.wait(1.0);
memory.store(new Chunk("cat"));
```

and run the following code on another agent thread:

```
agent.wait(2.0);
Chunk chunk = memory.recall(new Query("cat"));
```

The first thread requests that time advance to 1.0 seconds, while the second thread requests an advance to 2.0 seconds. In this situation, the first thread succeeds in advancing the clock to 1.0 seconds, and then performs the memory store. The second thread's *wait* step will block until 2.0 simulated seconds have passed, guaranteeing that the subsequent *recall* step will happen only after the first thread's *store* step.

The underlying implementation of the shared clock uses a type of cyclic barrier (Java's Phaser class) to ensure that all concurrent threads are synchronized as each reaches a new time step. If desired, the system can be run in real time such that the clock corresponds to the actual time (or a multiple thereof). In typical usage, however, simulated time does not need to correspond to real time; in fact, it is often advantageous to run the simulation as quickly as possible, and then examining time after the fact for various purposes (e.g., to predict how long a particular set of actions might take).

Memory System

The memory system is based wholly on the ACT-R theory of declarative memory (Anderson, 2007). In this theory, memory consists of *chunks* of knowledge with slot-value pairs, which over time strengthen or decay with practice or lack of use. In particular, each chunk has an associated *activation* that defines how readily the chunk can be recalled, as dependent on its prior usage: if a chunk is "used" (recalled or re-stored) at times t_k, the activation A_i for chunk i can be defined as

$$A_i = ln\left(\sum_k t_k^{-d}\right)$$

where d is the memory decay factor that determines how quickly the usages decay. Given the chunk's activation, we can determine the probability to recall the chunk as

$$Pr\ (recall) = \frac{1}{e^{-(A_i-\tau)/s}}$$

for a retrieval threshold τ and noise level s. If the chunk can be successfully retrieved, the time needed for retrieval is defined as

$$T_{recall} = Fe^{-A_i}$$

In our cognitive code system, chunks are stored using the `memory.store()` action, which incurs only a 50 ms time for a cognitive step—the same 50 ms delay used in many cognitive architectures for the firing time for a production rule (e.g., ACT-R, Soar). For recall, our system includes two types of actions. First, there is a non-blocking action `memory.startRecall()` that initiates the recall action but allows the code to continue past this point after a 50 ms cognitive step time. A non-blocking action of this type allows the code to continue and perform other actions while recall is taking place (e.g., watching for a visual stimulus or typing a key). The `memory.getRecalled()` action is then used to access the recalled information, and this command blocks until the recall is complete. Second, there is the blocking command we saw earlier, `memory.recall()`, which is equivalent to performing a `memory.startRecall()` followed by a `memory.getRecalled()`. Both types of recall actions take as an argument a query that partially defines the desired chunk (as seen earlier with the "cat" example), and both can fail and return *null* if the chunk is not successfully recalled.

Chunk rehearsal—that is, the usages defined earlier—can take several forms. When a chunk is initially created, this is defined as its first use. If an exact copy of this chunk is stored later, the copy is merged into the original chunk and becomes another use of the chunk. Finally, any recall of the chunk serves as a rehearsal and adds to the use count. Thus, all of these forms contribute to the gradual increase in a chunk's activation; in contrast, the lack of use causes any early uses to decay away, making the chunk more difficult to recall and more costly (in terms of time) if successfully recalled.

Perceptual System

The perceptual system is primarily based on ACT-R and partly based on a related theory of eye movements. The vision module, following ACT-R, assumes a spotlight of visual attention that moves according to a two-stage *where-what* process of finding objects and encoding objects. The non-blocking action `vision.startFind()` attempts to find a visual location that matches the given query based on perceptual features available in peripheral vision (like position, color, etc.), and its complementary command `vision.getFound()` returns the found location. Their blocking counterpart `vision.find()` achieves the same effect in a shorthand simpler action. Another command, `vision.waitFor()`, waits until a location matching the query appears in view (e.g., to model waiting for a visual stimulus). Once a location is found, there are analogous actions for encoding the object at the location and returning information about the object: the non-blocking action `vision.startEncode()` and its associated action `vision.startEncode()`, along with their blocking counterpart `vision.encode()`.

The movement of visual attention from one object to another generates movements of the system's simulated eyes as defined by the EMMA theory (Salvucci, 2001), which in turns derives from the E-Z Reader theory of eye movements in reading (Reichle, Pollatsek, Fisher, & Rayner, 1998). This provides the system with a powerful predictive dimension: the code never explicitly moves the eyes, but in moving attention, the eyes follow and demonstrate several interesting aspects of eye-movement behavior: the time lag between a movement of attention and a movement of the eyes; the possibility of skipping over an encoded object when that object is easy to encode (e.g., a high-frequency word); and the possibility of re-fixating an encoded object with multiple eye fixations when that object is difficult to encode (e.g., a low-frequency word).

In addition to vision, the system includes audition to model detection and encoding of aural information. Specifically, the audition module includes `audition.startDetect()` and `audition.detect()` actions (non-blocking and blocking, respectively) to detect a sound, and an associated action `audition.waitFor()`that waits for the next sound that matches a query. It also includes `audition.startEncode()` and `audition.encode()` actions that are analogous to these actions in the vision module.

Motor System

The motor system is currently focused on using a mouse and keyboard in a desktop computer environment. The mouse-movement module uses Fitts' law in the same manner as ACT-R and EPIC (Meyer & Kieras, 1997). The module includes the expected actions to move and click the mouse: `mouse.startMoveTo()` and `mouse.startClick()` as non-blocking actions, and `mouse.moveTo()` and `mouse.click()` as blocking actions.

For typing, the motor system is based on the TYPIST model (John, 1996). Typing is invoked with a `typing.type()` action that specifies the text to output. The typing module breaks up the given text into words and then types each word as a 50 ms cognitive step followed by the execution of the motor actions for each keystroke; shifted keys (e.g., capital letters) require a keystroke for the *shift* key before the keystroke for the character. The time for each keystroke was estimated as a function of typing speed as measured in gross words per minute. Specifically, a function was fit to Figure 4 of John (1996) to yield the following estimate of keystroke time T_{key} as a function of words per minute wpm:

$$T_{key} = .0000083(wpm)^2 - .003051(wpm) + .31727$$

The typing module can thus be set to any typing speed between 30 and 120 words per minute for an estimate of typing times at that speed.

```
[1]   String word = (String) vision.encode(vision.waitFor(new Query("word")));
[2]   Chunk chunk = memory.recall(new Query("pair").add("word", word));
[3]   if (chunk != null)
[4]       typing.type(chunk.getString("digit"));
[5]   int digit = (Integer) vision.encode(vision.waitFor(new Query("digit")));
[6]   memory.store(new Chunk("pair").set("word", word).set("digit", digit));
```

Figure 1: Cognitive code for the paired-associates task.

The other component of the motor system is the speech module, as represented by a `speech.say()` action that takes a simple string as input. Roughly like the ACT-R speech system, this module breaks the string into syllables and outputs the speech with a delay equal to a base time (200 ms) plus an execution time per syllable (150 ms). Whereas ACT-R has a simple assumption of syllables (one syllable per 3 characters), the module here uses a more complex method to break up syllables according to a number of rules for English pronunciation tested on a small corpus of common words.

Cognitive Code as Software

The fact that cognitive code is implemented as software in a mainstream programming language lends it several benefits over cognitive models developed in a typical cognitive architecture approach. In contrast to a monolithic cognitive architecture, modules in cognitive code are instantiated as needed, and using different implementations of a particular type of module does not pose a problem. For example, the following code defines a new agent, the eyes of the agent, and finally the vision module:

```
Agent agent = new Agent();
Eyes eyes = new Eyes(agent);
Vision vision = new Vision(agent, eyes);
```

Various methods for these components are easily accessible: for instance, a programmer might check aspects of the agent (e.g., `agent.getTime()`), move the eyes to a new location (`eyes.moveTo()`), or change parameters of the visual system (`vision.setFindTime()`). Developers can extend objects to include additional functions, or can build new objects that use cognitive-code objects as primitives. In fact, as the approach evolves, we expect different implementations of the modules to provide alternative theoretical approaches—in this case, say, we might have a different visual system based on pixels and salient features.

Another large benefit is that cognitive code inherits the many structures and tools already used by modern software for developing, interfacing, and testing code. Instead of having specialized IDEs (integrated development environments), cognitive code allows a programmer to use their preferred IDE for development. Cognitive code also inherits the robust APIs (application program interfaces) of modern programming languages, such as packages, classes, interfaces, and related constructs—a big advantage in accessing others' code and successfully integrating it with one's own code. Finally, cognitive code can be tested rigorously utilizing the same tools commonly in use today (e.g., JUnit tests in Java). In this case, each test can check not only whether the code runs correctly as a piece of software, but also whether some cognitive code fits an appropriate empirical (human) data set; in other words, the code's correctness also depends on whether it accurately mimics the behavior of human behavior in the chosen domain.

Illustrative Examples

We now provide a few illustrative examples of cognitive code, all of which represent re-implementations of existing models developed in a cognitive architecture. Our goal here is to demonstrate that cognitive code can produce much the same behavior and predictions as architecture models, but the code blocks that generate these behaviors are simpler and more learnable than their architectural counterparts.

Paired Associates

One of the standard models in Unit 4 of the ACT-R tutorial[2] is a model of the paired-associates task. In this task, participants see a word stimulus (e.g., "king") and must type a digit that is associated to that word in the experiment (e.g., "7"). The participant does not know the associations at the outset, but over time, they learn them and gradually become better at recalling the associated digit, improving their correctness and (for correct responses) improving their response times. Students studying the ACT-R architecture might model this task as they learn to understand ACT-R memory theory and implement their first models of memory storage and recall. Even after a few prior lessons in the syntax and semantics of the ACT-R modeling language, the paired-associates model can be difficult for students to understand and might take a typical student one to a few hours to work through and understand.

A cognitive-code model of the paired-associates task is shown in Figure 1. Starting at line 1, the model first waits for a word, blocking on the `vision.waitFor()` command until the stimulus appears, and then encodes the word. It then tries to recall a chunk that represents the word-digit pair in memory; if the chunk is successfully recalled, the model types the digit as a response. The model then waits for the digit (which appears in all cases) and stores the word-digit pair to memory. (Note that if the word-digit pair is already in memory, this strengthens the pair chunk as described earlier.) This model behaves essentially the same as the ACT-R tutorial model, and successfully produces the behavioral

[2] http://act-r.psy.cmu.edu

```
[1]   Visual visual = vision.find(new Query("word"));
[2]   while (visual != null) {
[3]       vision.encode(visual);
[4]       agent.wait(MEMORY_RECALL_DURATION);
[5]       visual = vision.find(new Query("word").add(Visual.SEEN, false));
[6]   }
```

Figure 2: Cognitive code for the reading task.

```
agent.run(() -> {
    Object tone = audition.encode(audition.waitFor(new Query("tone")));
    if (tone.equals("low"))
        speech.say("low");
});
agent.run(() -> {
    Object stimulus = vision.encode(vision.waitFor(new Query("stimulus")));
    if (stimulus.equals("O--"))
        typing.type("1");
});
agent.wait(1.0);
vision.add(new Visual("stimulus", 10, 10, 10, 10), "O--");
audition.add(new Aural("tone"), "low");
agent.waitForAll();
```

Figure 3: Cognitive code for the dual-choice task: blue for the aural-vocal task, green for the visual-manual task.

```
0.000   agent              wait for {isa=stimulus}
0.050   agent              wait for {isa=tone}
1.000   agent.vision       found {isa:"stimulus" x:10 y:10 w:10 h:10 seen:false}
1.000   agent              encode {isa:"stimulus" x:10 y:10 w:10 h:10 seen:false}
1.050   agent.audition     found {isa:"tone" heard:false}
1.050   agent              encode {isa:"tone" heard:false}
1.185   agent.vision       encoded O--
1.185   agent              type "1"
1.235   agent.hands        typing "1"
1.385   agent.audition     encoded low
1.385   agent              say "low"
1.435   agent.speech       saying "low"
1.444   agent.hands        typed 1
1.785   agent.speech       said "low"
```

Figure 4: Trace of the cognitive code in Figure 3: blue for the aural-vocal task, green for the visual-manual task.

patterns exhibited by people, namely the increased accuracy and decreased response time with practice.

Upon learning the cognitive-code approach, arguably the most difficult aspect of this code is learning the way that timing works—especially understanding that some actions will block until a stimulus appears or until a chunk is recalled. In general, though, a programmer versed in Java can easily understand the control flow here, and knows how to get this code to compile correctly and how to access API documentation when needed. (For example, we have not explained the details of the *Query* class used in the visual and memory requests, but these details are easily discovered in the documentation through a modern IDE.)

Reading

As part of a validation of the EMMA model of eye movements, Salvucci (2001) described a parsimonious model of sentence reading that simply encoded words from left to right (ignoring any deeper understanding to focus on the eye movements themselves). This model was a test of EMMA's ability to predict eye movements directly from straightforward shifts of visual attention, examining measures of gaze durations, first-fixation durations, and skip probabilities as a function of word frequency.

A similarly straightforward snippet of cognitive code that performs sentence reading is shown in Figure 2. The code implements a loop that iteratively finds and encodes each word. The find actions in lines 1 and 5 utilize the property of the visual system that, by default, vision finds locations closest to the current eye location; in line 5, the find command also makes sure to find a word that has yet to be seen. When a "visual" is found, the model encodes the contents of the word and fakes the semantic processing of the word by simply waiting for some time delay intended to mimic a lexical retrieval. Of course, a more rigorous model of reading would need to flesh out this aspect of the code, but for now, this code is sufficient to move attention along from one word to the next, triggering the predictions of the EMMA model and its resulting eye movements. The behavior of this model fits well to the empirical data, with correlations above .95 and low errors for the three measures mentioned above.

When developing such a model in a production-system architecture, the control flow of the model can be very difficult for students to grasp—both the flow within an individual production rule and the higher-level control flow among the production rules. In contrast, the iterative loop here is a familiar construct to programmers, and more clearly demonstrates the simplicity of the reading model and thus the predictive power of the underlying model of eye movements.

Dual Choice

Human multitasking has been characterized as the interaction of separate *cognitive threads* that interleave their processing (Salvucci & Taatgen, 2008, 2011). Figure 3 shows a simple example of how threading would work in a cognitive-code approach, illustrating a model of a dual-choice task that has been characterized as "perfect time-sharing" (Schumacher et al., 2001). This code starts two threads (via the `agent.run()` command): the first thread (lines 2-4, blue text) listens for a tone and then generates a speech response; and the second thread (lines 7-9, green text) waits for a visual stimulus and then generates a keystroke response. The task code (lines 11-14) waits 1 second, presents simultaneous visual and aural stimuli, and finally waits for all threads to complete.

The resulting simulation trace of this model, including timing (left-most column), is shown in Figure 4, with trace events color-coded as belonging to the first thread (blue) or the second (green). Both threads start waiting for their respective stimuli at the outset of the simulation. When the stimuli appear at the 1.0-second mark, the second thread sees the visual stimulus and begins to encode it; meanwhile, the first thread requires 50 ms to detect the aural stimulus, and after this delay it also encodes the sound. The motor responses—typing for the second thread, speech for the first—overlap such that neither thread experiences any time delays. Thus, the overall trace closely resembles the kind of perfect time-sharing behavior exhibited by more complex ACT-R models of this task (e.g., Salvucci & Taatgen, 2008).

Discussion

Cognitive code aims to strike a balance between the theoretical rigor of modern cognitive architectures and the practicality of modern programming languages and environments. The above examples show how concepts of cognitive code can lead to much simpler models, especially when compared with production-system architectures, and especially for the typical programmer versed in procedural and object-oriented languages commonly in use today.

There are at least two limitations of cognitive code compared to production systems that should be noted. First, production systems have the potential to be more flexible in their flow of control, whereas the procedural code here has a more rigid sequencing of actions. However, one might argue that most production-system models do not exploit this flexibility, but instead constrain the rules to embody the same kind of procedural control flow as the cognitive code here. Second, production systems allow for learning of the rules themselves (e.g., ACT-R's production compilation), whereas cognitive code is fixed by the developer. Allowing for procedural learning through cognitive code is still under exploration, but for now, this is perhaps its biggest limitation compared to production systems. Nevertheless, we remain hopeful that the benefits of the cognitive-code approach will ultimately pay dividends in expanding the usability and learnability of cognitive modeling to a wider audience.

References

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

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

Anderson, J. R., Bothell, D., Lebiere, C., & Matessa, M. (1998). An integrated theory of list memory. *Journal of Memory and Language*, *38*, 341-380.

Borst, J.P., Taatgen, N.A., & van Rijn, H. (2010). The problem state: A cognitive bottleneck in multitasking. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, *36*, 363-382.

Gray, W. D. (2008). Cognitive architectures: Choreographing the dance of mental operations with the task environment. *Human Factors*, *50*, 497-505.

John, B. E. (1996). TYPIST: A theory of performance in skilled typing. *Human-computer interaction*, *11*, 321- 355.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: Part 1. Basic mechanisms. *Psychological Review*, *104*, 3-65.

Laird, J. E. (2002). Research in human-level AI using computer games. *Communications of the ACM*, *45*, 32-35.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1-64.

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

Reichle, E. D., Pollatsek, A., Fisher, D. L., & Rayner, K. (1998). Toward a model of eye movement control in reading. *Psychological Review*, *105*, 125-157.

Salvucci, D. D. (2001). An integrated model of eye movements and visual encoding. *Cognitive Systems Research*, *1*, 201-220.

Salvucci, D. D. (2006). Modeling driver behavior in a cognitive architecture. *Human Factors*, *48*, 362-380.

Salvucci, D. D., & Taatgen, N. A. (2008). Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, *115*, 101-130.

Salvucci, D. D., & Taatgen, N. A. (2011). *The Multitasking Mind*. New York: Oxford University Press.

Schumacher, E. H., et al. (2001). Virtually perfect time sharing in dual-task perfor- mance: Uncorking the central cognitive bottleneck. *Psychological Science, 12*, 101–108.

Taatgen, N. A., Huss, D., Dickison, D. & Anderson, J. R. (2008). The acquisition of robust and flexible cognitive skills. *Journal of Experimental Psychology: General*, *137*, 548-565.