SIMCog-JS: Simplified Interfacing for Modeling Cognition - JavaScript

Tim Halverson (thalverson@gmail.com)
Oregon Research in Cognitive Applications, LLC
Oregon City, OR 97045 USA

Brad Reynolds (reynolds.157@wright.edu)
College of Engineering and Computer Science, Wright State University
Dayton, OH 45435 USA

Leslie Blaha (leslie.blaha@us.af.mil)
711th Human Performance Wing, Air Force Research Laboratory
WPAFB, OH 45433 USA

Abstract

A continuing hurdle in the cognitive modeling of human-computer interaction is the difficulty with allowing models to interact with the same interfaces as the user. Multiple attempts have been made to add this functionality (e.g., Hope, Schoelles, & Gray, 2014) in limited domains. This paper presents a solution allowing models to interact with web browser-based software, while requiring little modification to the task code. Simplified Interfacing for Modeling Cognition - JavaScript (SIMCog-JS) allows the modeler to specify how elements in the interface are translated into ACT-R chunks, allows keyboard and mouse interaction with JavaScript code, and allows sending ACT-R commands from the external software (e.g., to add instructions). The benefits, drawbacks, and future functionality of SIMCog-JS are discussed.

Keywords: Cognitive Architectures; Task Interface; ACT-R; WebSockets; JSON; HTML; JavaScript; D3

Introduction

A substantial challenge with modeling human cognition is the presentation of task environments to the simulated human. Software re-implementation provides little scientific reward, yet modelers face this burden every time they utilize a new or modified task. The situation is further complicated if a modeler is studying human-computer interaction (HCI) with complex software in which users are engaging in ongoing, dynamic, and interleaved or multi-tasking behaviors. Because the focus of cognitive modeling in HCI is often either explaining or predicting performance differences between alternative interfaces, substantial research time is spent re-implementing multiple, complex interfaces; this effort is further multiplied if multiple cognitive architectures are used.

Although re-implementation within a modeling architecture framework can allow maximum control by the modeler, it introduces additional challenges: (a) Re-implementation increases the likelihood that the fidelity of the simulation is degraded by an imperfect porting of the user interface or task dynamics. (b) Iterative changes to the original software/task require additional efforts to integrate these changes into the model's task environment. (c) Task-simulation environments for cognitive architectures are sometimes written in programming languages not commonly used for building HCI interfaces (e.g., ACT-R uses Lisp; Anderson et al., 2004) and often provide limited facilities for building the task simulations. Thus, the process of re-implementation forces a trade-off between task fidelity and time savings. An alternative to re-implementation is to allow a model to communicate directly with a user interface that is external to the cognitive architecture. Previous research has attempted to solve this challenge, although in limited domains. Computer vision (CV) has been used to automatically extract relevant visual features from an existing computer interface (e.g., Halbrügge, 2013; St Amant, Riedl, Ritter, & Reifers, 2005). While CV solutions remove the burden of "translating" the interface to symbols understood by the architecture, they also reduce the control the modeler has on how the visual interfaces are specified. Additional control requires the modeler to customize the CV algorithms or specify screen element "templates" at the pixel level. Other solutions provide the ability for models to act within specialized environments, like games (e.g., Veksler, 2009) or robotics (e.g., Kennedy, Bugajska, Adams, Schultz, & Trafton, 2008). These solutions are incredibly useful but are limited to their specialized environments. Still other solutions provide a more general framework for interfacing models with external software by using interprocess communication protocols available in many programming languages (e.g., Büttner, 2010; Hope et al., 2014). The solution presented herein falls into this final category.

We present a solution to the challenge of communication between external task environments and cognitive architectures: Simplified Interfacing for Modeling Cognition - JavaScript (SIMCog-JS). Our approach supports communication between Java ACT-R (Salvucci, 2013) and HTML-/JavaScript-based software in a user-friendly manner. In the remainder of this article, we specify some design requirements, describe the functionality provided by SIMCog-JS, and provide an example of SIMCog-JS applied to a dynamic, multitasking experiment environment.

SIMCog-JS Design Requirements

SIMCog-JS is a technology that allows cognitive modelers to specify how visual information is extracted from external software, passes that information to ACT-R, and passes keyboard and mouse events back to the external software. The primary motivation for SIMCog-JS is rooted in a desire to

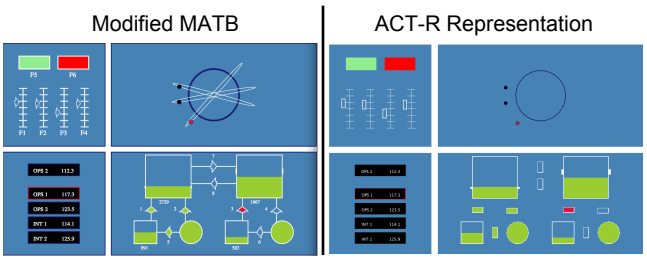

Figure 1: The browser-based modified Multi-Attribute Task Battery (mMATB) as it would appear to a human participant (left) and a representation of the ACT-R visicon (right).

apply cognitive architectures to dynamic, multitasking experiments, such as training simulations or naturalistic web-browsing. We desire a flexible system allowing interaction between multiple cognitive modeling formalisms and existing software/HCI environments.

SIMCog-JS attempts to minimize the modeler's burden in multiple ways. First, SIMCog-JS requires minimal modification of existing task code, although it does require that the modeler have access to the JavaScript task code. Second, SIMCog-JS includes an extension to Java ACT-R that replaces Java ACT-R task code and requires no modifications for a wide variety of tasks. Third, SIMCog-JS provides a user-friendly, flexible syntax for specifying which visual elements should be passed to ACT-R, when those elements should be updated in ACT-R, and how they should be perceived (i.e., slot values).

In order to provide this functionality, SIMCog-JS had three critical design requirements:

1. SIMCog-JS must use standard software protocols for communication between models and experimental software.
2. Integrating model interactions with the task makes minimal modifications to the experimental code, minimizing interference with human data collection or natural behaviors.
3. Model execution occurs in real time.[1]

We note that as our initial target task environment, the modified Multi-Attribute Task Battery (mMATB), executes in a web-browser and the modeling formalism, Java ACT-R, is written in Java, we were required to implement a new solution to facilitate interaction between cognitive models and a task environment. Hope et al. (2014) introduced a similar solution for interfacing Lisp ACT-R with stand-alone software. However, that published solution does not support either Java or JavaScript. Our solution took motivation from Hope et al.'s work.

The mMATB Task Environment

We apply SIMCog-JS to a dynamic, multitasking environment, mMATB (Cline, Arendt, Geiselman, & Blaha, 2014). This is a generalized version of the MATB developed to assess multitasking in pilot-like environments (Arnegard & Comstock, 1991); the modifications in this environment make similar cognitive demands on the participants, but the tasks are less pilot-specific in nature. Our browser-based implementation is written with the D3 JavaScript library (Bostock, Ogievetsky, & Heer, 2011) integrated with a Python django database. Participants interact with the environment through keyboard button presses and mouse clicks and movements.

The mMATB, shown in the left panel of Figure 1, entails four separate tasks, which we summarize clockwise from the upper left. The upper left quadrant is a Monitoring Task, consisting of a set of sliders and two color indicator blocks. The participant's task is to provide the appropriate button press (F1-F6, labeled on each indicator/slider) if a parameter is out of its normal state. For the sliders, this means moving above or below ± 1 notch from the center. For the indicators, the normally green (black) might turn black (red).

A Tracking Task is contained in the upper right quadrant, wherein three colored circles move continuously along individual ellipsoid trajectories. At any time, one of the circles may turn red, indicating it is the object to be tracked by the participant. The participant tracks the target by mousing to the target, clicking on it, and then following it with the mouse, until the next target object is indicated with a color change.

The lower right quadrant contains a Resource Management Task. Two resource tanks are schematically illustrated, together with representations of fuel sources, reserve tanks, and gated connections (each numbered 1-8) between all tanks. The participant's task is to maintain the resource levels within a range specified by bars on the sides of the tanks. The on/off states of the gates are controlled with number pad key presses. The participant can control the gates with any strategy of choice to maintain the resource levels.

Finally, the lower left quadrant contains a Communications Task. The display shows four channels (Int1, Int2, Ops1, Ops2) together with the current channel values; the topmost line gives a target channel and value. If a red cued target appears in the top box, the participant uses the up/down arrow keys to select the cued channel and the right/left arrow keys to adjust the channel value to the new cued value. The enter key submits the corrected channel, which changes the topmost cue box to white until the next channel cue appears.

Cognitive modeling of mMATB performance aims to capture behavioral impacts of changes in workload, operator stress levels, or fatigue levels and to characterize the high-level strategies engaged during continuous multitasking.

SIMCog-JS Software Architecture

SIMCog-JS uses a client-server software architecture. The server exists within Java ACT-R (Salvucci, 2013) as a "generic task." This generic task is populated with environment-specific information as the server receives messages from a client describing the current state of a task interface. The server dynamically changes the ACT-R envi-

[1] As our target software does not support synchronized execution with external software. This is not a constraint unique to our target software, as web browsers (and most software) do not allow external synchronization.

ronment based on the messages received from the client and sends messages to the client describing ACT-R's actions.

The client is built in JavaScript, allowing it to run within all modern web browsers. The client runs alongside the browser-based task translating the task interface for the server and processing interactions from ACT-R. The client is integrated into existing code by referencing the client script in the task's primary web page (e.g., index.hmtl). Further, the modeler specifies three things within the client: (a) a list of visual chunks to be represented in ACT-R, (b) a list of ACT-R commands for the model, and (c) handlers for interactions received from the task.[2] Once these are in place, the system is ready for use.

The client and server communicate via WebSockets and JavaScript Object Notation-Remote Procedure Call (JSON-RPC). WebSockets (Fette & Melnikov, 2011) allow reliable, simultaneous connections between the client and server.[3] Once connected, the client and server use JSON-RPC (JSON-RPC Working Group, 2010) to send information. JSON-RPC is a standardized protocol for sending messages based on the JSON standard. Both the WebSocket and JSON-RPC protocols are standards that have been implemented in many programming languages, allowing SIMCog to be easily extended to task interfaces and cognitive modeling formalisms in other programming languages through the use of these standard protocols and reuse of the SIMCog-JS's messaging specification. WebSockets and JSON are native to JavaScript, but requires additional libraries for Java.

Figure 2 shows the flow of information between the browser task environment (i.e., client), the server, and ACT-R. After Java ACT-R and the client are configured and running, the client sends all information about the interface to the server at the start of the task, along with any initial ACT-R commands. As keyboard and mouse events are generated by ACT-R, these actions are passed to the client to affect the interface. Details on how to configure the client and server can be found with the SIMCog-JS Software Design Document included in SIMCog-JS distribution.[4] The following section provides details on how this communication takes place between the client and ACT-R.

Communicating through SIMCog-JS

SIMCog-JS allows the modeler to specify how interface elements in software will be represented in ACT-R, to send ACT-R commands from the task client to Java ACT-R, and to determine how the task client will respond to keypresses and cursor movements made by the model. The following sections describe how these three facilities are used.

[2]As discussed in Keypress and Mouse Events section below, default keypress and mouse click handlers are provided for the modeler's convenience.

[3]The client and server may be run on separate computers and over the internet. However, doing so may introduce additional lag that could reduce the fidelity of the simulation.

[4]The SIMCog-JS distribution can be downloaded from: http://sai.mindmodeling.org/simcog/

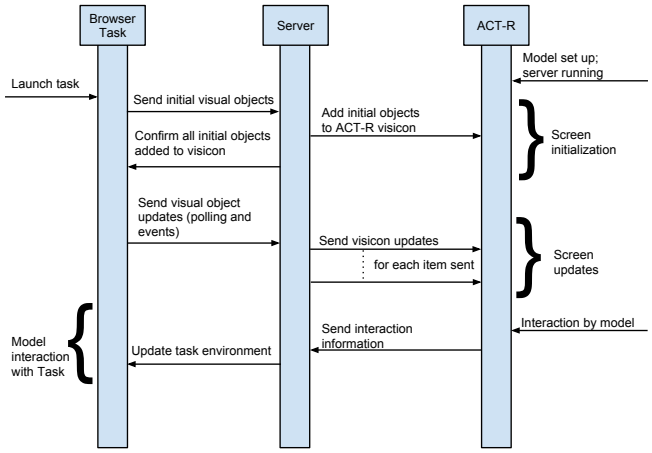

Figure 2: Information flow through SIMCog-JS. The information events start with the modeler initiating the Java ACT-R model and then launching the browser-based task. SIMCog-JS then connects the two environments. The vertical dimension captures time flowing from top to bottom.

Specifying Visual Chunks

The left side of Figure 1 shows the visual interface for mMATB task as presented to the human participant in the web browser. The right side of Figure 1 shows a visual representation of ACT-R's visicon, as specified by the modeler and displayed by the SIMCog-JS server. The modeler specifies which web-browser elements become visual chunks in ACT-R, how those elements will be represented in ACT-R, and when those elements will be updated. SIMCog-JS does not send all interface elements to ACT-R; doing so could unnecessarily complicate the modeling. The modeler may have observational data or theoretical reasons for hypothesizing that some interface elements are completely ignored by users. For example, a uniform background frame may have no impact on performance, assuming adequate contrast between the background and foreground elements. Therefore, the modeler must specify the set of interface elements that become visual chunks in ACT-R.

The modeler must specify the interface element `id` and the element's shape `type`. The object's coordinates, width, height, color, and text (if applicable) are automatically extracted from the task interface using JavaScript DOM function calls and jQuery-dependent CSS specificity computations.[5] The syntax for specifying visual objects is:[6]

[5]All attributes can be specified manually. See the Design Document at http://sai.mindmodeling.org/simcog/ for more information and useful links to the libraries utilized.

[6]All syntax descriptions follow the same convention. Angle brackets are used to indicate a value that must be specified by the modeler. Values enclosed in quotation marks indicate that the value is a string. For example, id:"<unique_name>" indicates that unique_name should be replaced by a string that is the value of the id, like id:"foo". Alternative values are separated by "|".

```
{id:"<unique_name>", type:"<valid_type>"}
```

The id uniquely identifies the object. If the interface element has an explicitly labeled id in the document-object model (DOM; e.g.,), that string can be used as the SIMCog-JS id. If an object does not have a unique ID, the id can be specified with two attributes, name and domLocation. The name value is a string that must be unique to the object. It is helpful to make the name meaningful. A domLocation value is the node of the object located within the DOM tree. This node can be found in multiple ways. One way is to identify the relation to another named object within the DOM tree. Another is to locate the object in the DOM tree relative to the root (i.e., document). Syntax for these methods are:

```
{id:{domLocation:document.getElementById(
    "<element_id>").nextElementSibling,
  name:"<unique_name>"}, ...}

{id:{domLocation:document.body.
    lastElementChild,
  name:"<unique_name>"}, ...}
```

The type is a string that determines how the object will be represented within ACT-R. A screen object must be one of nine types: "Line", "Cross", "Label", "Oval", "OvalOutline", "OvalOutlineFill", "Rectangle", "RectangleOutline", or "RectangleOutlineFill". The first three types are "native" to Java ACT-R;[7] the remaining items are custom task components added by the authors. The type of an object is represented in the visual chunk's "isa" attribute (e.g., "OvalOutlineFill" has an attribute of "isa oval"). Additionally, if the shape is specified with two colors (e.g., "OvalOutlineFill" has a fill and outline color), then SIMCog-JS adds a borderColor chunk slot that contains the value of the border's color and the standard ACT-R color slot contains the value of the fill color. The coordinates, dimensions, and colors of objects are determined differently for different object types. If an object is declared with the wrong type, it is likely that the object will be misrepresented in ACT-R.

The modeler may also specify when changes to interface elements are sent to ACT-R. The default is to update whenever the element changes using DOM Mutation Observers. This event-based functionality is most useful when one or more attributes of the interface element changes infrequently. The modeler may also specify that updates occur at a configurable, regular interval (e.g., polling). This polling functionality is most useful when the attributes of objects are rapidly changing. In such cases, the polling method can substantially decrease the number of messages to the server, decreasing computational demands. Specifying polling-based changes is done by adding a change attribute with the value "poll" to the element declaration. Finally, an object can be declared as static. Static elements are never updated. The modeler may specify an object as static by adding a change attribute with the value "static" to the element declaration. Syntax for change declarations is:

```
{..., change:"evt"|"poll"|"static"}
```

In addition to specifying when updates for an object are sent, the modeler may specify which visual properties are updated. By default, all properties are updated. Listing only those properties that will change can improve software performance. For example, a light may only change color but not move, or tracking reticles may only change coordinates but not colors. The list of properties that will be updated are appended to the value given to the change attribute. If no such list is given, all properties are updated. Valid attributes are "x", "y", "height", "width", "color", "secondaryColor", and "stringVal". Only labels have "stringVal" attributes. Syntax of these expanded change declarations are:

```
{..., change:["<attribute_name>",
    "additional_attribute_name>, ...]}
{..., change:["poll", "<attribute_name>",
    "additional_attribute_name>, ...]}
```

It is also possible to add objects to the ACT-R task environment that are not relevant to the model but are useful for the modeler (i.e., for debugging the visual interface). This is done using "task-irrelevant" objects. Task-irrelevant objects never appear in the model's visicon. For example, a task-irrelevant object may be used as a background to make objects easier to see for the modeler. There are four possible task-irrelevant objects: Cross, Label, Line, and Rectangle. Task-irrelevant objects are not updated throughout the task. All objects default to being task-relevant. To declare an object as task-irrelevant, the attribute taskRelevant is added to an object declaration with a value of false. The syntax for this option is:

```
{..., taskRelevant:true|false}
```

Example Specifications from mMATB This section provides examples of how interface elements in the mMATB task, shown in Figure 1, are specified. The examples start with simple specifications and progress to the more complex.

Perhaps the simplest interface elements in mMATB are the background color panels underlying all four quadrants. They never change (i.e., are static), are filled with a single color ("steel blue"), and are rectangular. If one hypothesizes that these background colors are ignored by the users, these elements can be declared as task-irrelevant. Alternatively, the cognitive model could simply ignore these elements, or the modeler could choose to exclude these elements. Making them task-irrelevant will improve software performance ever so slightly. Including them in the interface specification will make the interface in ACT-R more readable. Although the interface element is simple, it is not uncommon for HTML ids to be missing from background elements, which complicates the id for these elements. In this example, the domLocation value is used to determine the id based on the modeler's knowledge of the location of these elements in the DOM tree.

[7]Note that "Button"s are not supported. As discussed later, any type of object can be clickable.

```
{type:"Rectangle",
  id:{name:"svg0",
    domLocation:d3.
      selectAll("svg")[0][0].firstChild},
  change:"static",
  taskRelevant:false}
```

The Monitoring Task color indicator blocks (upper left quadrant) provide a straightforward example for displaying event-based task elements. The following example is the specification of the green color indicator block; the specification of the red block is similar. The id of this rectangular element is known, monitor_button_0. The only property that changes is the color and so the only value assigned to the change attribute is color. The changes are infrequent, normally changing only a few times per second, so the change attribute is given the value of "evt". Note that "evt" is the default and is not required in the declaration.

```
{type:"Rectangle",
  id:"monitor_button_0",
  change:["evt", "color"]}
```

Label interface elements are unique in that they contain text that can be updated. The mMATB Communications Task's channel values provide examples of changing labels. As with the indicator blocks, the ids are known, like "comm_channel_1_frequency" in the example below. However the text of the label changes. In the example below, the change attribute is labeled as event-based (e.g., "evt") because the values rarely change, and only the text of the label is marked for change with " stringval ".

```
{type:"Label",
  id:"comm_channel_1_frequency",
  change:["evt","stringVal"]}
```

The most dynamic elements in the mMATB interface are the colored circles in the Tracking Task. Each oval moves continuously along a path using the D3 animation library. The constant motion produces a lot of events; this could generate a lot of network traffic and decrease software performance. Therefore, these elements are specified with the "poll" value for the change attribute. The location ("x" and "y") and "color" change, and so all three values are listed in the change attribute. The final attribute of the example specification given below is clickable ; this attribute will be described in the next section.

```
{type:"OvalOutlineFill",
  id:"track_circle_0",
  change:["poll","x","y","color"],
  clickAble:true}
```

Keypress and Mouse Events

To complete the interaction loop, actions taken by the model are transmitted to the task environment. There are three types of interaction currently supported by SIMCog-JS: key press, cursor move, and mouse click. The server sends all interactions to the client; the modeler has full control of how to handle (or ignore) events.

The simplest of the three interactions is key press. Key press interactions are handled automatically by the system. This is done by mapping ACT-R keycodes to JavaScript keycodes and dispatching a keydown event to the task. Currently only keydown events are supported; the modeler may modify the client code to support keyup and keypress events.

When a click is performed, a message is sent to the client containing the location of the mouse and the event type (mouseClick). While mouse coordinates may be enough for many tasks, more information is provided, for example, to deal with the asynchronous nature of the system or facilitate a deeper analysis. An example from the mMATB task is when the model clicks on circles in the tracking task that are moving quickly; the circle could move a couple of pixels out from under the cursor before the click event reaches the client. To handle such circumstances, objects can be declared as clickable . Anytime a click is performed by the model, the server determines if the click was performed within any of the clickable objects. If it is determined that one or more objects were clicked, the message to the client will also include the unique IDs of the items clicked, along with the location, type, and ID of every clickable object. This information allows for cases where the unique identifier is needed to click an object within the task and even more complex cases where specific information and computation is desired.

To declare a visual chunk as clickable, add the clickable attribute to an object's specification and set it to true.

```
{..., clickable:true}
```

The client automatically handles clicks by dispatching a JavaScript mouse click event. If a clickable object was clicked, the client dispatches a click event for that element. Otherwise, the client finds the element at the location of the click and simulates the click there.

For mouse movements, JavaScript does not allow control of the cursor in web browsers. Such control is not allowed by code in web browsers for security and usability reasons. To simulate a model's mouse movements in the task, SIMCog-JS generates mouse movement messages for the client. This approach offers both reliability and speed without introducing external software systems.

When the model moves its simulated mouse, a mouseMove message is sent to the client that contains the location of the model's simulated cursor. With this information, the modeler can record the simulated mouse movements similarly to how human mouse movement data are recorded. To do so, the modeler will likely need to modify the client code. For example, in mMATB the cursor-recording code looks like:

```
ws.onmessage = function (evt) {
    //Called when server message received
    var serverMessage = JSON.parse(evt.data);
  ...
  else if(serverMessage.Command == "mouseMove"){
    track_chart.mouseLocation(
      {x:modelInteraction.mouseX,
      y:modelInteraction.mouseY});
    }}
```

Sending ACT-R Commands

SIMCog-JS supports sending model commands from the task to the model. Doing so is straightforward and takes advantage of existing Java ACT-R methods for executing ACT-R commands. The modeler adds commands to a list in the client code that is sent to the server at the start of execution. For example, to represent the Resource Management Task instructions to maintain the resource level within a target range, the modeler may specify:

```
["(add-dm (resourceTask isa goal
      minLevel 2000 maxLevel 3000))",
 "(goal-focus resourceTask)"]
```

Conclusion and Future Work

SIMCog-JS is a system that allows cognitive models to interact with external software, minimizing the task re-implementation burden on the modeler. The system currently facilitates communication between Java ACT-R and HTML/JavaScript. In addition to describing the architecture of SIMCog-JS, this paper reported on using SIMCog-JS to (a) specify visual interface elements for use by ACT-R and how those interface specifications can be customized, (b) integrate ACT-R responses into JavaScript software, and (c) execute ACT-R commands from the task interface. The strengths of SIMCog-JS are the easy specification of visual objects and interactions with minimal task-code modifications and the seamless interaction between models and browser-based tasks. The modeler need only specify the identity and shape for visual objects to reach ACT-R.

Development is ongoing to improve and extend the functionality of SIMCog-JS. A mid-term goal is to add synchronous execution modes, where the task and model use the same simulation clock, relaxing design requirement 3 without negatively impacting real-time execution. Additional planned features include audio event specification and support for multiple cognitive modeling formalisms, like EPIC architecture (Kieras & Meyer, 1997) and Python-based mathematical models.

By harnessing standard programming protocols and languages, the SIMCog approach can lighten the modeler's burden while broadening the environments in which computational cognitive models operate. Because SIMCog-JS can operate in an environment with facilities for complex data visualization (e.g., D3), we will be pushed to enhance ACT-R's functionality. In the future SIMCog-JS could be integrated with an artificial vision system to, for example, automatically determine object shape; this combined approach could, in fact, bolster both candidate solutions to the task re-implementation challenge.

Acknowledgments

This research was supported AFOSR. Distribution A: Approved for public release; distribution unlimited. 88ABW Cleared 12/16/2014; 88ABW-2014-5938.

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004, October). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036–1060.

Arnegard, R. J., & Comstock, J. R. (1991, May). Multi-attribute task battery: Applications in pilot workload and strategic behavior research. In *6th international symposium on aviation psychology* (pp. 1118–1123). Columbus, Ohio.

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3: Data driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 2301–2309.

Büttner, P. (2010). "Hello Java!" Linking ACT-R 6 with a Java simulation. In D. D. Salvucci & G. Gunzelmann (Eds.), *International conference on cognitive modeling* (pp. 289–290). Philadelphia, PA.

Cline, J., Arendt, D. L., Geiselman, E. E., & Blaha, L. M. (2014, May). Web-based implementation of the modified multi-attribute task battery. In *4th annual midwestern cognitive science conference.* Dayton, Ohio.

Fette, I., & Melnikov, A. (2011, November). *The WebSocket Protocol* (Tech. Rep. No. RCF 6455). Internet Engineering Task Force.

Halbrügge, M. (2013). ACT-CV: Bridging the Gap between Cognitive Models and the Outer World. In E. Brandenburg, L. Doria, A. Gross, T. Güntzler, & H. Smieszek (Eds.), *Berliner werk- statt mensch-maschine-systeme* (pp. 205–210). Berlin.

Hope, R. M., Schoelles, M. J., & Gray, W. D. (2014). Simplifying the interaction between cognitive models and task environments with the json network interface. *Behavior Research Methods*, *46*, 1007–1012.

JSON-RPC Working Group. (2010, March). *JSON-RPC 2.0 Specification.* http://www.jsonrpc.org/specification/.

Kennedy, W. G., Bugajska, M. D., Adams, W., Schultz, A. C., & Trafton, J. G. (2008). Incorporating Mental Simulation for a More Effective Robotic Teammate. In *Conference on artificial intelligence* (pp. 1300–1305). Chicago, IL.

Kieras, D. E., & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, *12*(4), 391–438.

Salvucci, D. D. (2013, August). ACT-R: The Java Simulation & Development Environment. *http://cog.cs.drexel.edu/act-r/index.html*.

St Amant, R., Riedl, M. O., Ritter, F. E., & Reifers, A. L. (2005). Image Processing in Cognitive Models with SegMan. In *Human-computer interaction international* (pp. 1869:1–1869:19).

Veksler, V. D. (2009). Second Life as a Simulation Environment: Rich, high-fidelity world, minus the hassles. In *International conference on cognitive modeling.* Manchester, United Kingdom.