SIMCog-JS User and Programming Manual

AFRL/711 HPW/RHCV

Document written by Brad Reynolds

Mentors: Leslie Blaha, Tim Halverson

Table Of Contents

1. Introduction

1.1. Purpose

The goal of this SIMCog development is to create a system that allows humans and cognitive models to perform the same task separately or in parallel. This allows for models to run a task, and for model-based predictions of human actions. Our program provides a streamlined interface between web browser-based tasks and a Java-based version of the cognitive architecture Adaptive Control of Thought-Rational (ACT-R). Although we use ACT-R for illustrative purposes, our program could be generalized to any cognitive architecture written in any language that supports standard networking and messaging libraries. Previous attempts to allow cognitive architectures to operate alongside humans were time consuming and fraught with logistical issues because the human user and cognitive architecture often performed the tasks developed in different frameworks. While these tasks written in different languages may be similar, they may not be exactly the same. With SIMCog-JS a modeler would only have to program a task once, using rapid development languages like HTML, CSS, and JavaScript. The system described in this document allows a Java ACT-R model to interact with the same task.

1.2. Reference Materials

JSONRPC2.0 -	Specifications	http://www.jsonrpc.org/specification
	Library	http://software.dzhuvinov.com/json-rpc-2.0.html
Java ACT-R		http://cog.cs.drexel.edu/act-r/
		General ACT-R information
Original NASA MATB		http://matb.larc.nasa.gov/
Java WebSocket		https://github.com/TooTallNate/Java-WebSocket
		http://java-websocket.org

1.3. Useful definitions

- **Interface Elements** – Elements within the web-based task environment. Each Interface element corresponds to a specific DOM tree node.
- **Visual Chunks** – Representation of interface elements within ACT-R, what the model will "see" in the visicon.
- **Screen Objects** – Declaration within SIMCog-JS that is used to translate information from Interface Elements to Visual Chunks.
- **DOM** – Document Object Model – Internal structure of a web page that is interpreted into a visual representation by the browser. The DOM is structured as a tree with each node being an element.

- **CSS** – Cascading Style Sheets – Style sheet format for describing the look and spacing of elements in a web page.
- **Specificity** – Web browsers way of determining which CSS styles should be applied to an element.
- **Cognitive Architecture** – A theory that outlines a way of artificially modeling human cognition and an implementation of that theory in a computer-programming framework, (e.g., ACT-R). Cognitive models may be built within a cognitive architecture. In this research we are using Java ACT-R as a computational formalism.
- **Cognitive Model** – Computational or mathematical formalism for representing or describing cognitive, perceptual, and motoric processes.
- **Visicon** – A list of objects that an ACT-R model sees and can respond to.
- **Hash Table** – Data structure used for storing visual chunk data server side. Uses a key-pair mapping of IDs to Visual Objects.

1.4. Required Software

SIMCog-JS Server and Client	http://sai.mindmodeling.org/simcog
JQuery (See "Color" in Section 7)	http://www.jquery.com/download/
Java SE Development Kit 7	Oracle Java 7 JDK
Java SE Development Kit 8	Oracle Java 8 JDK
Google Chrome	http://www.google.com/chrome/
Eclipse	http://www.eclipse.org

2. System Overview

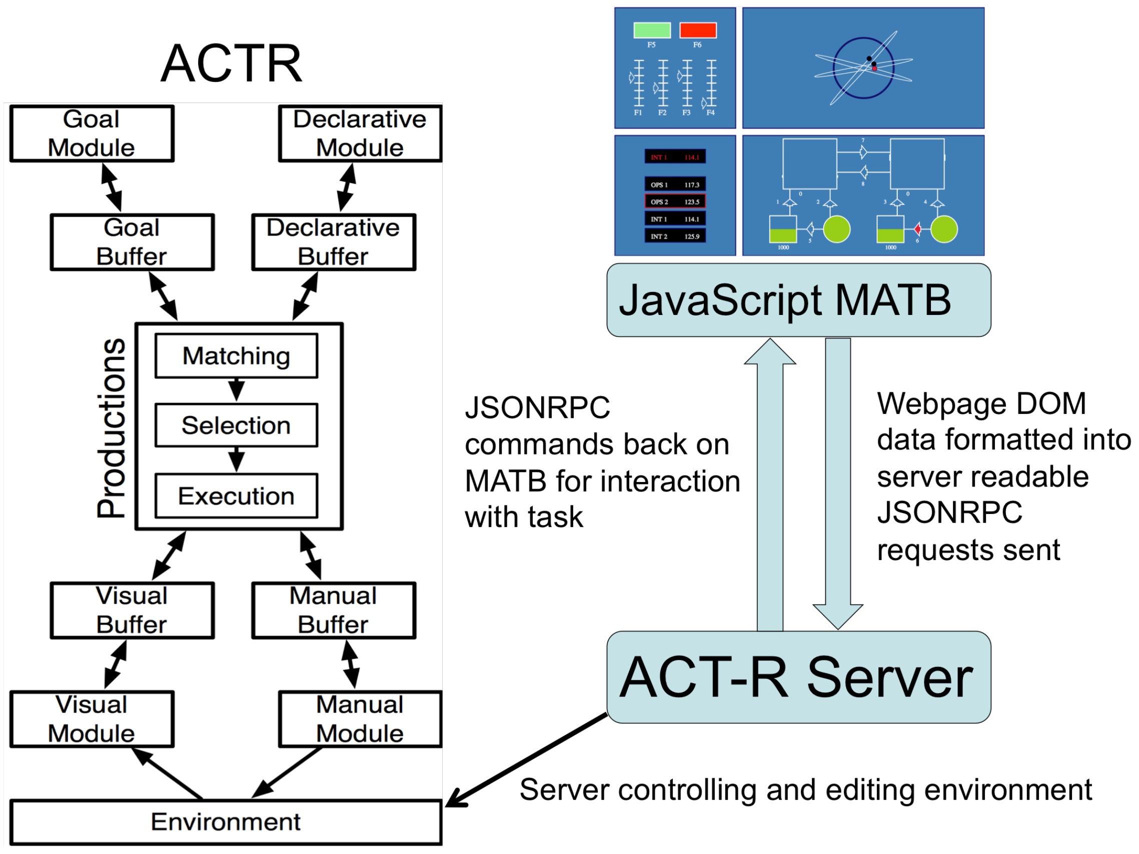
Simplified Interfacing for Modeling Cognition – JavaScript (SIMCog-JS) uses a client-server software architecture. The client is written in JavaScript and executes alongside the task. This client JavaScript file is external to the task files and is included by referencing it in the index page of the task's HTML file. While the client is mostly task-independent, some task-specific modifications are required.

The server is written in Java in order to work with the Java version of ACT-R. The server receives messages from the client and presents the appropriate information to ACT-R as well as sending interactions from the model back to the client. The server is not task-specific.

The system uses JSON-RPC to send messages between the server and client. There are other candidate messaging protocols such as XML-RPC. However, JSON was selected because it is native to JavaScript. The system uses WebSockets to connect the client and server. The WebSocket protocol use TCP connections and is native to JavaScript. On the Java side, the Java WebSocket library implementation of the WebSocket protocol is used.

For a connection to be established, the server must be running and waiting for a client. When the browser page containing the task is loaded, the JavaScript client makes the connection using WebSockets.

The full system running with the example experiment environment, the modified Multi-Attribute Task Battery (mMATB), can be represented as follows:



3. JavaScript Client Overview

The client runs when the web page containing the task is loaded; the client then attempts to make a connection to the server. If the server is offline, the client outputs an error message, but does not affect the task execution. This allows the task to be used with humans or computational models, depending on the server status, with no changes to any task code.

When the connection is successful, the client processes the web-browser interface elements as specified by the modeler (see Section 7). The client extracts the relevant features of the interface elements using JavaScript functions. These features are serialized as strings and sent to the server. The client then waits until the server responds with a sync message. The sync message indicates that the server is ready to receive update messages from the client. When the client receives a sync command, it starts to send updated information about specified screen objects to the server.

The client maintains two lists of screen objects to be updated based on the screen objects' specified update method. The polling list contains screen objects that update at a regular time interval. The event-based list contains screen objects that update based on DOM object events. Objects that have been declared as static are not included in either list, and are only sent to the server once (in the initial message). See the "change" variable in Section 7 for information about updating methods.

Screen objects in the polling list are updated regularly with a timeout function. The default time interval is 1/60 of a second. The time interval can be modified to suit the particular task or modeling environment. During each interval, screen object values that have been changed are polled for their current information. The client sends this updated information to the server in one JSON-RPC message. The use of one message reduces network traffic, especially when objects are updating very quickly.

Screen objects within the event-based list have mutation observers assigned to all of their DOM objects. A mutation observer is a listener that is called anytime the DOM object changes attributes. When a screen object's listener is called, the object's updated values are sent to the server.

The client also handles interactions from the model. Supported interactions are key presses, mouse movements, and mouse clicks. Key presses and mouse clicks are handled automatically by generating JavaScript events. Mouse movements must be handled manually by the programmer.

When the client receives a key press, it determines the key code using a mapping of ACT-R key codes (see Appendix B) to JavaScript key codes. The client then calls a custom function named *simKeyEvent*. This function provides a workaround for a known bug[1] with Chromium-based browsers and their declaration of key events. This solution will work in all tested browsers (Firefox, Chrome, Safari).

When the client receives a mouse click interaction from the server, it creates a click event. If screen objects have been declared as "clickable" (see Section 7) and the server returns one or more of those clickable objects as the target of the click, the click event is generated on that object. If the server does not return a list of clicked objects, the client will find the DOM object at the location and generate a click event on that object.

JavaScript does not allow control of the cursor in web browsers, creating problems generating mouse movement actions. While "mouse over" events can be generated by the client, this will result in erroneous data since the location of the real mouse would be used by the mouse over event. Therefore, mouse movement interactions must be handled differently; the user will have to call the function that they use to log human data when a mouse movement event is received from the server using the model's cursor location data (sent in the mouse movement event) instead of the browser's cursor location data.

[1] https://bugs.webkit.org/show_bug.cgi?=id16735
https://code.google.com/p/chromium/issues/detail?id?=328753

NOTE: The code is thoroughly documented with comments about what each function is doing.

4. Java Server Information

The server is built within the Java ACT-R task environment. The server connection listener starts when the Java ACT-R is started. The server can currently interface with just one task at a time. The server is fairly generalizable and requires little-to-no modifications to implement different tasks.

Five libraries are used to implement the server:

1-2. jsonrpc2-base and jsonrpc2-server: parse messages, dispatch method calls;
3. json-smart-1.2: parses JSON objects and arrays;
4. Java-WebSocket-1.3.0: implements the WebSocket protocol used to connect the client and server;
5. Java ACT-R: implements the cognitive architecture.

The server interprets JSON-RPC messages from the client about the current status of the task and relays them to ACT-R environment. To complete the event loop, the server listens for model interactions and relays them to the client. The server handles visual chunks being added and changed dynamically by having a core Hash Table where visual chunks are stored. The keys of this Hash Table are the IDs given to the interface elements by the client, and the values are the associated task environment objects. When adding or updating visual chunks, the server uses one method, *cycleVisObjectsToMap.* This method has the following guidelines:

1. If the ID isn't in the Hash Table, create a new visual chunk based on the information provided in by the client. Add that item to the Hash Table as well as the model's screen.
2. If the ID is in the Hash Table, get the visual chunk with that ID, check its variables and change them if they don't match the provided variables' information. This allows for only specific traits to change (e.g.,, only updating color).

5. Message Passing Specifics

The system has five JSON-RPC requests from the client to the server: *runCommand, init, update, deleteObjects*, and *exit*. The first command, *runCommand,* sends ACT-R commands (e.g.. *"add-dm"*) to the model. The second command sent by the client is *init*. This command sends a list of screen objects with unique IDs and their initial values. The server parses and presents the screen objects to the model, and then the server sends a sync message back to the client. The sync message informs the client that the server is ready to receive update requests.

The *update* is the next command that executes; this sends a list of screen objects and their updated values. Screen objects in the updated list can only include those screen objects that were sent in the *init* command. Unlike *init*, the *update* command does not reply with a sync, which allows for faster communication. Update commands are handled by the server in the order they are sent.

The *deleteObjects* command is a function for deleting visual chunks from the screen. It receives a list of unique IDs and removes the corresponding visual chunks from the model's visicon. The last command, *exit*, stops the server from listening.

An important part of the system is interactions by the server back on the client. These interactions include key press, mouse move, mouse click, and sync. These interactions can happen at any point during execution and are dependent on what the model does.

For key presses, Java ACT-R has a listener called *typeKey* that is called with the parameter of the key pressed when the model presses a key. With this information the server handles all key presses very simply, sending all key presses to the client in a JSON object with the "Command" variable of "keypress" and "index" variable of the given parameter. For example, when a model presses the "up" arrow key, the command would look like:

```
{Command: "keypress", index: "up-arrow"}
```

This allows a great amount of generalizability by sending every key press, letting the client decide to handle or not handle the command and how to go about handling the command. Currently the client simply generates a key down event and dispatches it.

For model mouse (i.e., cursor) move interactions, there is a listener called *mouseMove*. The location the mouse moved to and the command (*mouseMove*) are pushed into a JSON object and sent to the client. For example,

```
{Command: "mouseMove", mouseX: 25, mouseY: 12}
```

As discussed above, mouse move interactions must be explicitly handled in the client due to web browser dependencies on real mouse location; however the information provided is straightforward to use.

For mouse click interactions, there is a listener called *clickMouse* that is called every time there is a mouse click by the model. Mouse click commands sent to the client have been set up to be flexible in that they require no modification for simple cases but plenty of additional capabilities for more verbose data collection and accuracy needs. This information could be used to determine the degree of error in the click location or the object's relation to other targets when it was clicked.

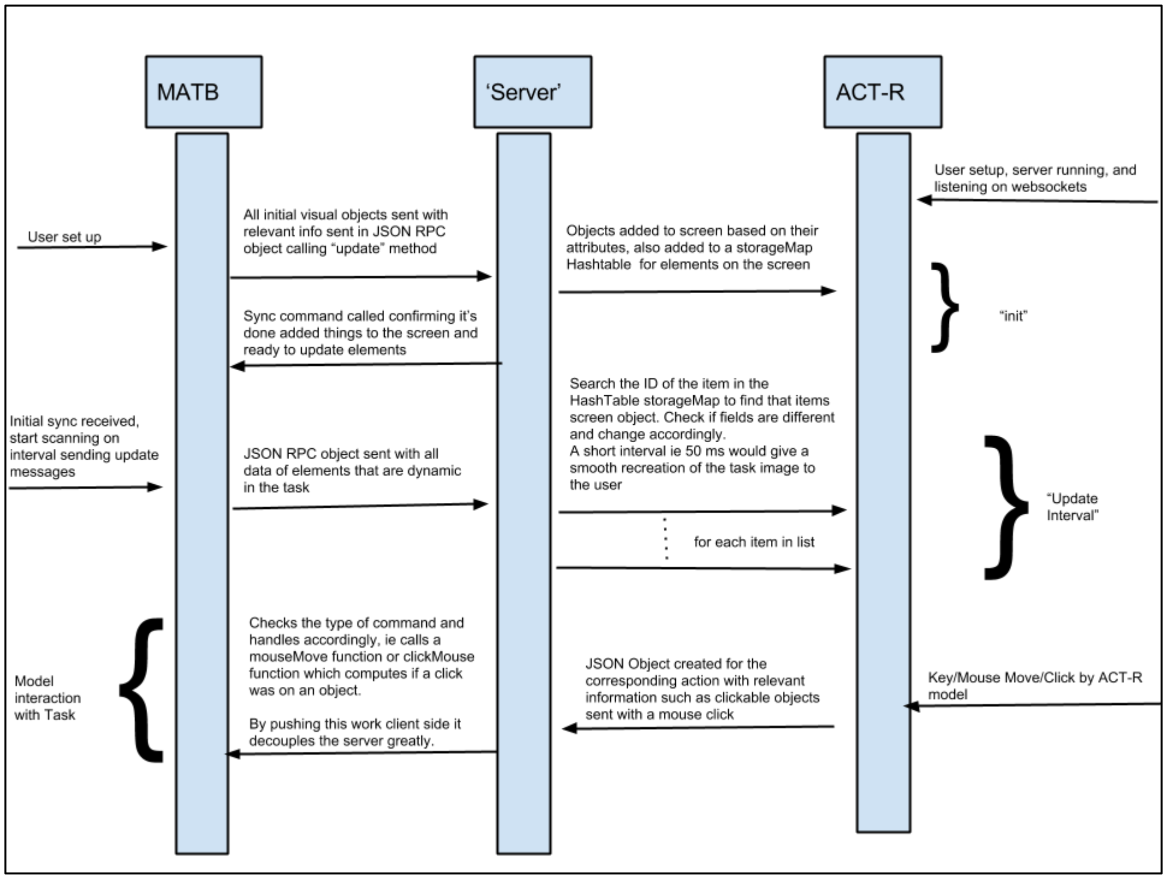
When screen objects are initially added (Server side in the *init* message), all screen objects with a variable *clickable* set to true are added to a list of clickable items. When the model issues a mouse click, the server determines which clickable screen objects were clicked based on the mouse click location and the location of all screen objects. The server then sends a JSON message that contains the click location, the command type (*mouseClick)*, a list containing the identifiers and locations of all clickable objects, and a list of identifiers for all clicked objects. In the cases where there are no clickable items or no clickable objects were clicked, the list will be empty.

A sample *mouseClick* command with two clickable objects, one of which has been clicked, will look like:

```
{
    "Command":"mouseClick","mouseX":38, "mouseY": 36,
    "allClickableObjects":
        [
         {"id":"circle1","objX":5,"height":4,"width":4"objY":5,"type":"Oval"},
         {"id":"circle2","objX":39,
         "height":10,"width":10,"objY":36,"type":"Oval"}
        ],

    "clickedObjectIdentifiers":
        [
            "circle2"
        ]

}
```

Mouse clicks are handled client-side by generating a key event on the items that were clicked. If the list of clicked items is empty, the client gets the interface element at the location of the click (if any) and generates a key event there. With the information provided, additional operations could be added client-side to log more data.

The overall dataflow outlined in this section can be seen in this diagram:


MATB
'Server'
ACT-R
User setup, server running, and listening on websockets
User set up
All initial visual objects sent with relevant info sent in JSON RPC object calling "update" method
Objects added to screen based on their attributes, also added to a storageMap Hashtable for elements on the screen
"init"
Sync command called confirming it's done added things to the screen and ready to update elements
Search the ID of the item in the HashTable storageMap to find that items screen object. Check if fields are different and change accordingly.
A short interval ie 50 ms would give a smooth recreation of the task image to the user
Initial sync received, start scanning on interval sending update messages
JSON RPC object sent with all data of elements that are dynamic in the task
"Update Interval"
for each item in list
Checks the type of command and handles accordingly, ie calls a mouseMove function or clickMouse function which computes if a click was on an object.
By pushing this work client side it decouples the server greatly.
Model interaction with Task
JSON Object created for the corresponding action with relevant information such as clickable objects sent with a mouse click
Key/Mouse Move/Click by ACT-R model

6. Setup and Basic Implementation

6.1 Generic setup instructions

A complete SIMCog-JS version can be downloaded at the SIMCog Project repository.[2] Included in this download is the client and server. The server can be found in the SIMCogServer folder; the server is saved as a zipped Eclipse project. The client can be found in the SIMCogClient folder and contains the two necessary files (the client and jQuery) and a zipped complete example.

To setup SIMCog with your task, follow the instructions listed below. For an in-depth example of setting up and running SIMCog with a task, see section 6.1 where the mMATB is used as an example.

1. Import the server as an Eclipse project
2. Run the project with actr.env.main as the "main class"
3. In the left console window, enter the text below.

   (set-task "actr.tasks.ActrInterface.SIMCogACTR")

4. Select "Run SIMCog Task" from the "Run" menu. The server is now running.

[2] http://sai.mindmodeling.org/simcog/index.html

5. Add the jquery-1.11.0.js and SIMCog.js to the folder where the index.html of the task is.
6. Edit the index.html of the task to include the two files by using the <script> syntax as seen below

<script src="./SIMCog.js"></script>
<script src="./jquery-1.11.0.js"></script>

7. Add interface elements that will be translated to ACTR in the screenObjects array at the top of SIMCog.js using the specified syntax in section 7 (Examples in section 6 as well)
8. Load the task webpage and the interface elements will be visible in ACT-R as specified.

Additional Modifications

1. A fully specified model, with production rules, parameter settings, etc., can be used to perform the task as would be done with any other model in Java ACT-R. See the Java ACT-R documentation for more information about using Java AC-TR.[3] Ensure that the set-task command at the top of the production file is the same as listed in Step 3 above.

2. To record mouse movement actions, the programmer must manually specify them. As explained in Section 5 this is because of browser limitations with controlling the mouse and generating mouse movements. This is straightforward to do. First, open SIMCog.js in a text editor and scroll down to ws.onmessage. This function is the 3rd function in SIMCog.js and is found under a block comment "Messages from Server". The programmer must edit the *mouseClick* block of code by making a call to the task's mouse movement data logging function. It is likely a call to this function would also be done in a web page mouse listener. The model mouse location will be the mouseX and mouseY attributes in the JSON object that has been received and parsed by the client.

3. To send ACT-R commands to the model when the server starts, the programmer can modify the two blocks of code. The first option is to edit the *modelCommands* array; this is the first element in the SIMCog.js file. This is the easiest approach and useful when the commands can be statically defined. Any number of commands can be added to this array. Alternatively, if there were a need to generate commands programmatically then the programmer would modify the *readyCommands* function. This is the first function in SIMCog.js. This approach is useful when you want to provide knowledge of where things are in the task to the model or generate commands dynamically based on task parameters. Two examples of defining ACT-R commands are:

```
var modelCommands = ["(add-dm foo)"]

var readyCommands = function(){
```

[3] http://cog.cs.drexel.edu/act-r/

```
if (taskSubsection.active == true){
      modelCommands.push("(add-dm bar)");
      }
}
```

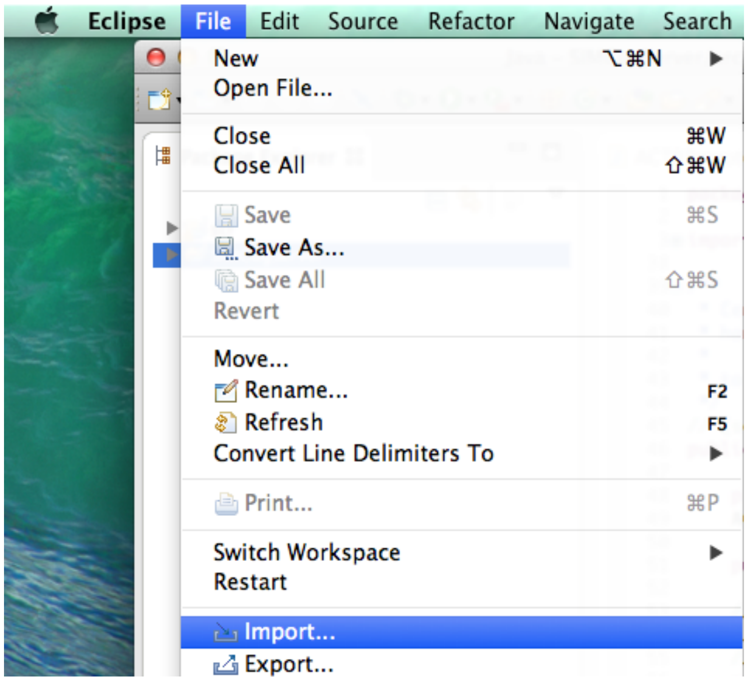
4. Just as in the standard LISP implementation of ACT-R, Java ACT-R allows the evaluation of custom LISP-like statements in a production rule. This is done by overriding methods in SIMCogACTR.java. Override the *public double bind(Iterator<String> it)* method for functions that return values to the right-hand-side of productions. Override the *public void eval(Iterator<String> it)* method for functions that do not return values to the right-hand-side of productions. Override the *public void evalCondition(Iterator<String> it)* method for functions that do not return values to the left-hand-side of productions.

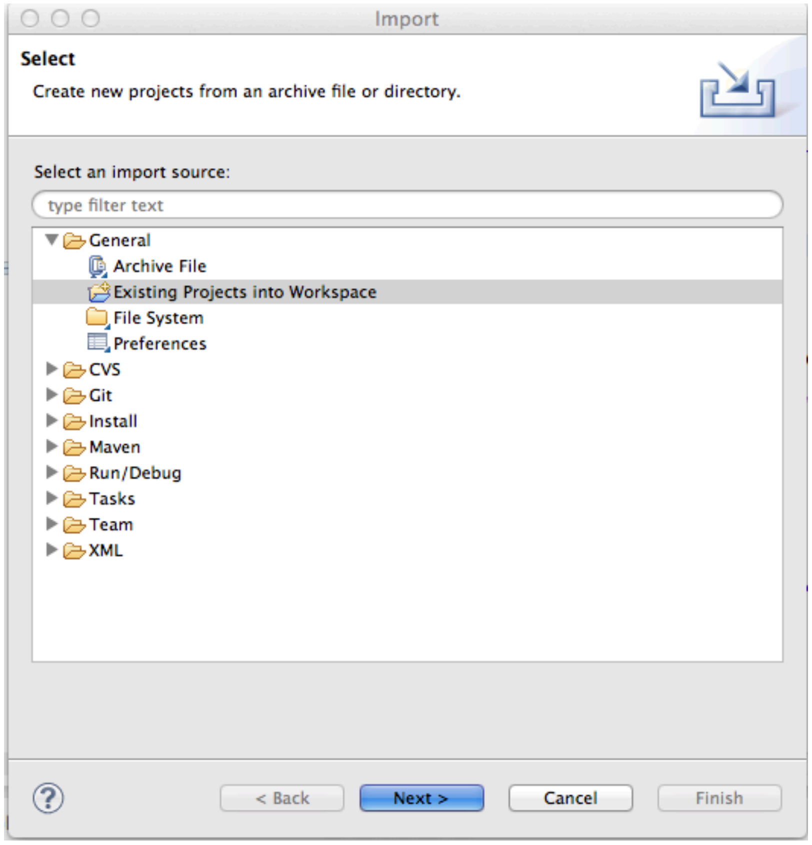
6.2 mMATB Sample Setup

This section will give an example of setting up a task based on the instructions in Section 6.1 using the mMATB as the task. All of the completed files from this example are included in the download from the SIMCog Project repository (Meaning no edits to code will need to be made)..

1) Importing the server – Click the "Import…" option under "File" in Eclipse. Select "Existing Projects into Workspace" (under the General options) and click the "Next" button. Set the "Select" option to "Select archive file", then select the project "SIMCogServer", and then press the "Finish" button.

Note: In the Eclipse Preferences, be sure to set the Installed JRE to a version of Java that is 1.7 or higher (This system was built using Java SE 7 [1.7.0_60] and tested in Java SE 8 [1.8.0_31]). Also be sure to set the Java Compiler Compliance Level to the corresponding version of Java (1.7 or 1.8) that you are using. See Required Software (Section 1.4) for a download link to Java SE 1.7.0_60 and Java SE 1.8.0_31.

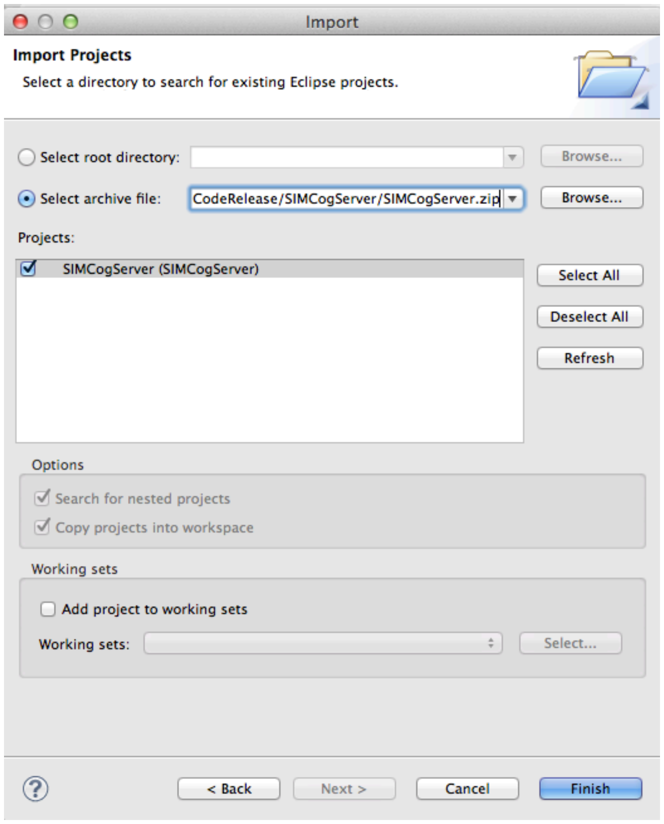



Import

Select

Create new projects from an archive file or directory.

Select an import source:


Import

Import Projects

Select a directory to search for existing Eclipse projects.

- [ ] Select root directory:
- [✓] Select archive file: CodeRelease/SIMCogServer/SIMCogServer.zip

Projects:

- [✓] SIMCogServer (SIMCogServer)

Options

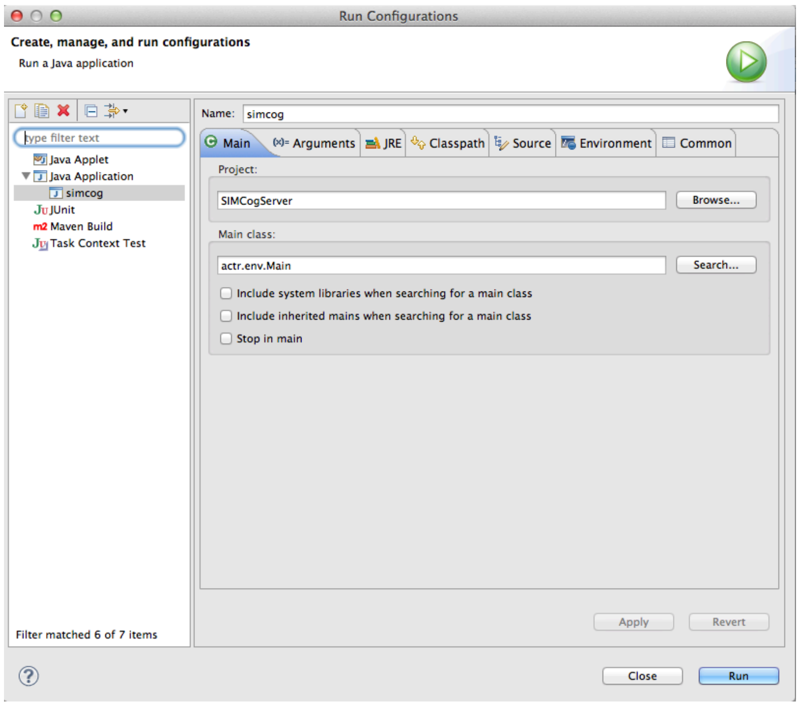
- [✓] Search for nested projects
- [✓] Copy projects into workspace

Working sets

- [ ] Add project to working sets

Working sets:

2) Run the project - Make a new run configuration by going to Run → Run Configurations. Select Java Application on the left-hand list and hit the "new" button (as indicated in the onscreen instructions). The project should be SimCogServer and the main class should be "actr.env.Main". You can type this class in, or search for the main class. When this new configuration is run, it will open the ACT-R window.


Run Configurations

Create, manage, and run configurations

Run a Java application

Name: simcog

Project: SIMCogServer

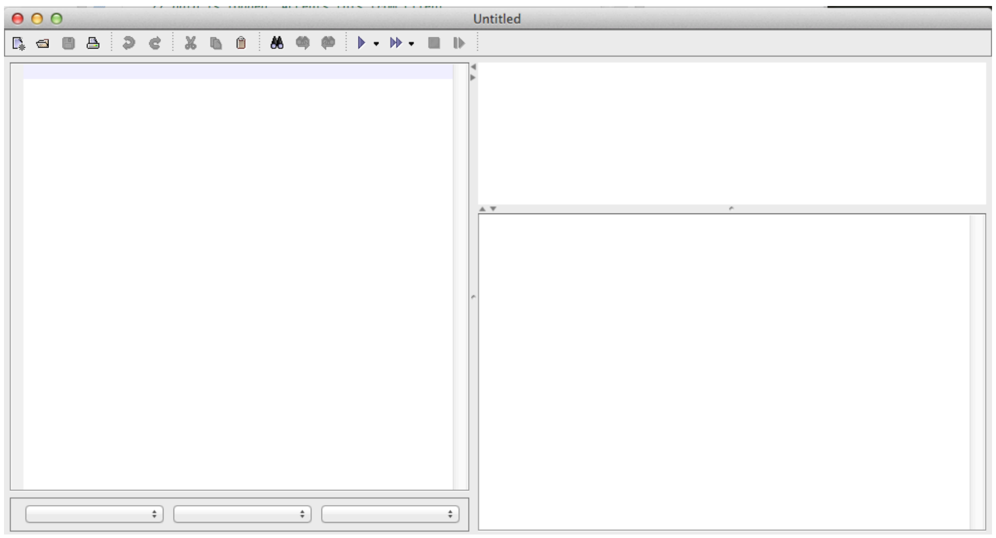
Main class: actr.env.Main

☐ Include system libraries when searching for a main class

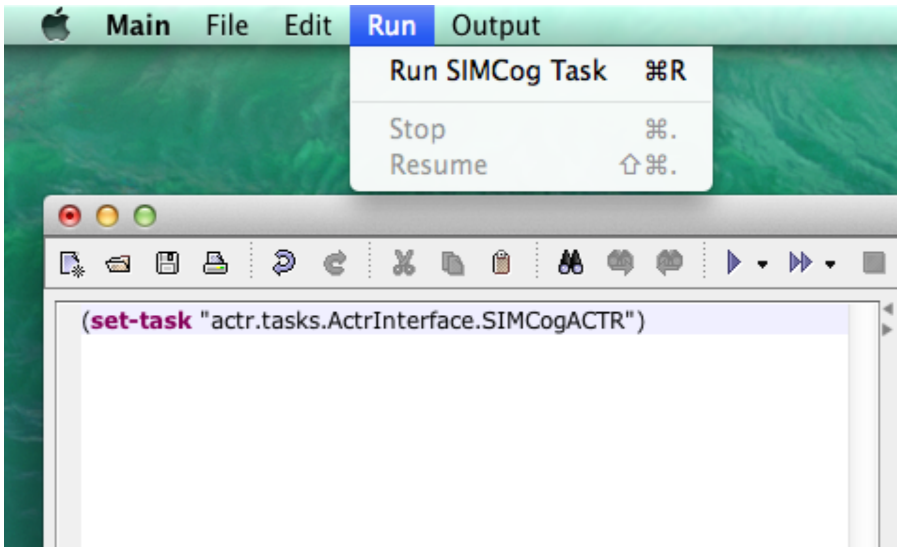
☐ Include inherited mains when searching for a main class

☐ Stop in main

ACT-R Window

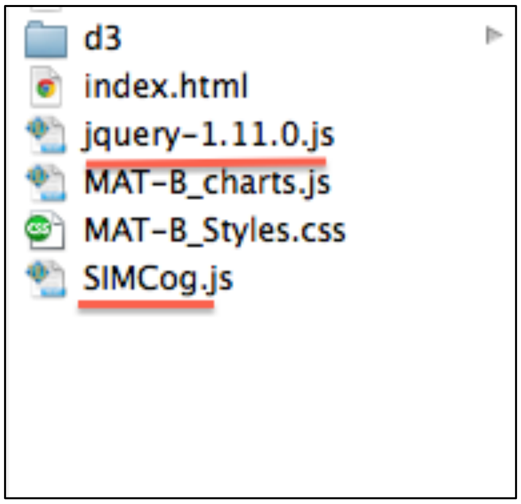


3) Starting the server listening. In the left panel of the ACT-R Window insert the start command **(set-task "actr.tasks.ActrInterface.SIMCogACTR")** in the left command window and select “Run SIMCog Task” from the “Run” menu. The server will be listening and ready for connections.



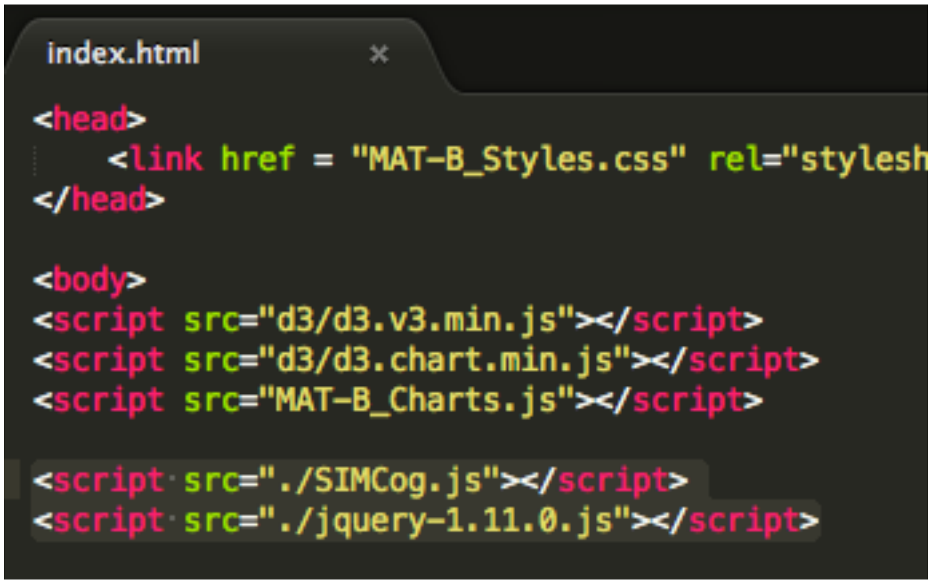
```
(set-task "actr.tasks.ActrInterface.SIMCogACTR")
```

4) Add the SIMCog.js and jQuery-1.11.0.js files to the directory of the task (in file system). Add the two JavaScript files to the directory containing the task's index.html file.



5) Add references to the two JavaScript files in index.html. The reference syntax is as follows

```
<script src="./SIMCog.js"></script>
<script src="./jquery-1.11.0.js"></script>
```

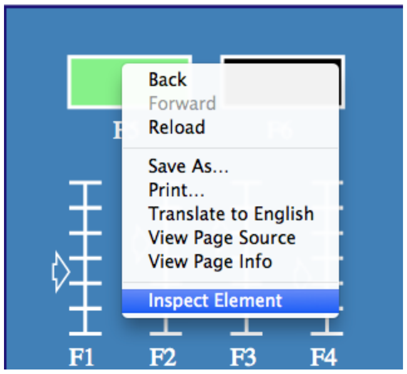


```
index.html
<head>
    <link href = "MAT-B_Styles.css" rel="stylesh
</head>

<body>
<script src="d3/d3.v3.min.js"></script>
<script src="d3/d3.chart.min.js"></script>
<script src="MAT-B_Charts.js"></script>

<script src="./SIMCog.js"></script>
<script src="./jquery-1.11.0.js"></script>
```

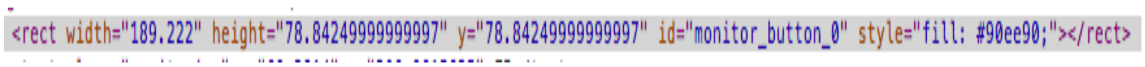
6) Specify screen objects within the task. Once steps 1-5 are complete the server and client can connect. Start the static mMATB by launching a local server in the directory. This can be done by opening a Terminal window, cd to the directory containing the task index.html, and using the command **python –m SimpleHTTPServer 8888** (be sure to type, not copy and paste, this command). Then open the web browser, and got to **localhost:8888 (http://127.0.0.1:8888)**.

To test the client-server connection if the task is functional, open up the web browser that is running the task and check the JavaScript console (View>Developer>JavaScript Console) and there will be an "Opened connection with ACT-R server" message. There won't be anything visible within the upper-right pane in ACT-R as there are no specified screen objects in the client yet.

To easily find the necessary details of an element to be specified, right click on the element in the browser and select "Inspect Element". Below the green button has been right clicked.



The developer tools window will open after selecting “Inspect Element.” This will be a representation of the DOM tree with the information about the selected element highlighted.

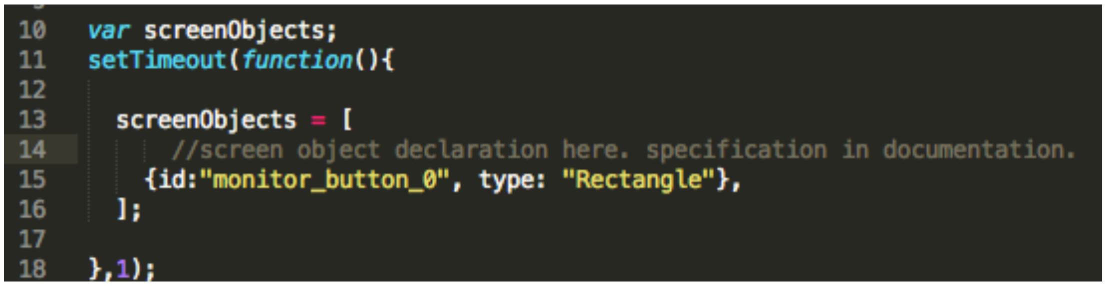


```
<rect width="189.222" height="78.84249999999997" y="78.84249999999997" id="monitor_button_0" style="fill: #90ee90;"></rect>
```

As can be seen, the green button has an ID. To add the green button as a screen object, create a JSON object containing its id and type, in this case it a simple Rectangle. The object looks like

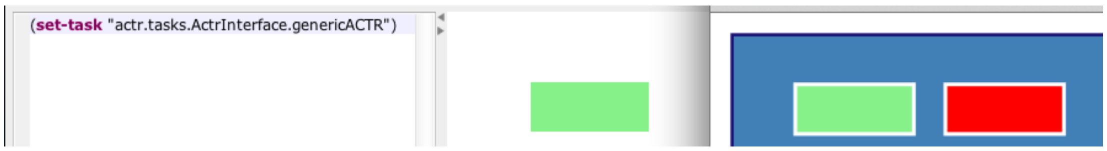
{id:”monitor_button_0”, type:”Rectangle”}

Add this object to the screenObjects array at the top of the SIMCog.js file like below.



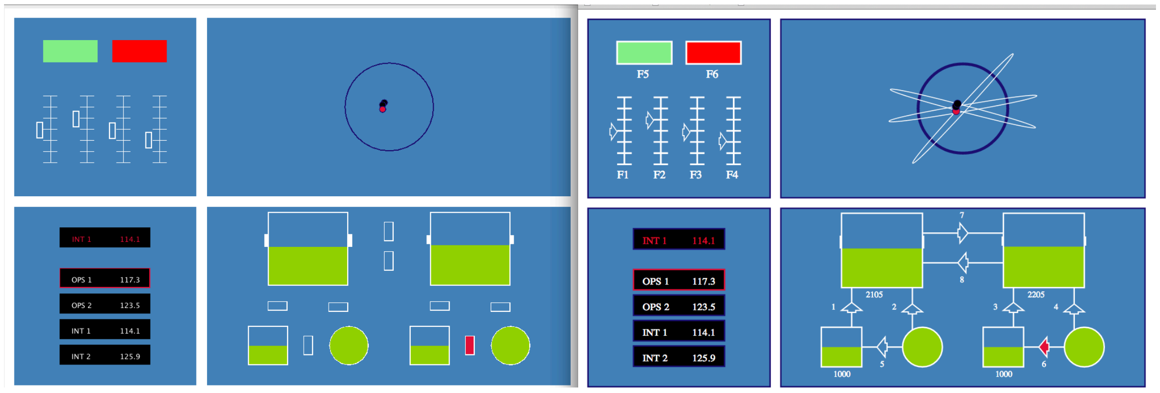
```
var screenObjects;
setTimeout(function(){

  screenObjects = [
    //screen object declaration here. specification in documentation.
    {id:"monitor_button_0", type: "Rectangle"},
  ];

},1);
```

When Java ACT-R is ran again and the task is reloaded, the button will appear in the upper-right pane of the Java ACT-R interface.



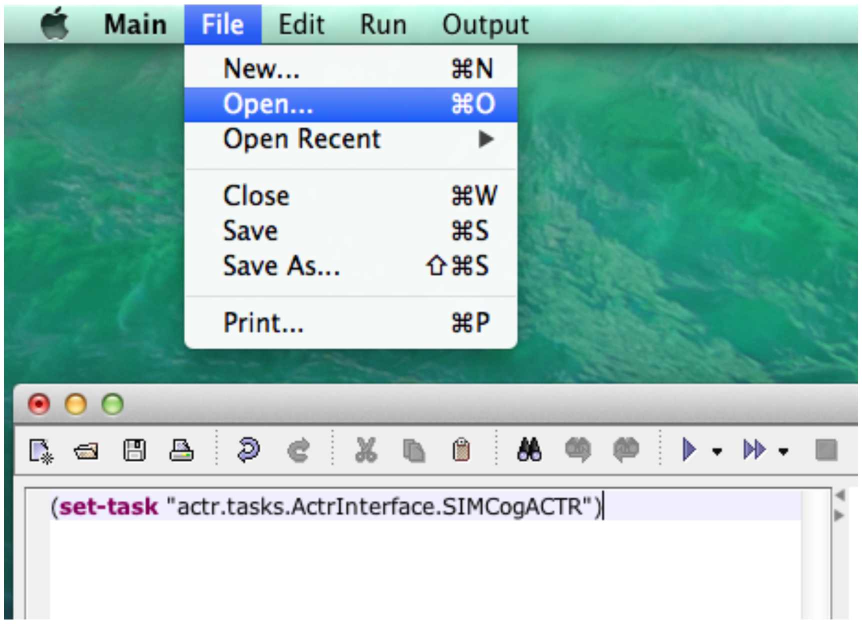
```
(set-task "actr.tasks.ActrInterface.genericACTR")
```

7) Load the browser page when completed - After all relevant interface elements in the modified MATB are transferred the translation will look like

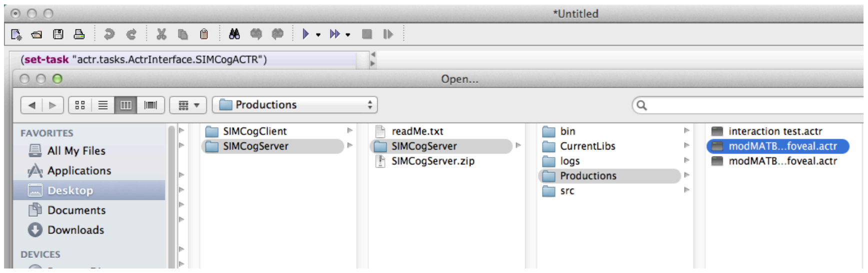


Additional Modifications

1. Running the model – Included in the server download is a folder called "Productions". This folder contains .actr files that are models for the mMATB. To run these restart the SIMCogServer killing the window and re-running the configuration you made. When the ACT-R window opens click File>Open in in ACT-R window and navigate to the mMATB production rules file (located in the Eclipse project in the Productions folder). Select the desired model file and click the "OK" button. Then run the model by selecting "Run SIMCog Task" from the "Run" menu. Reload the task window in the same was as before and the model will run.

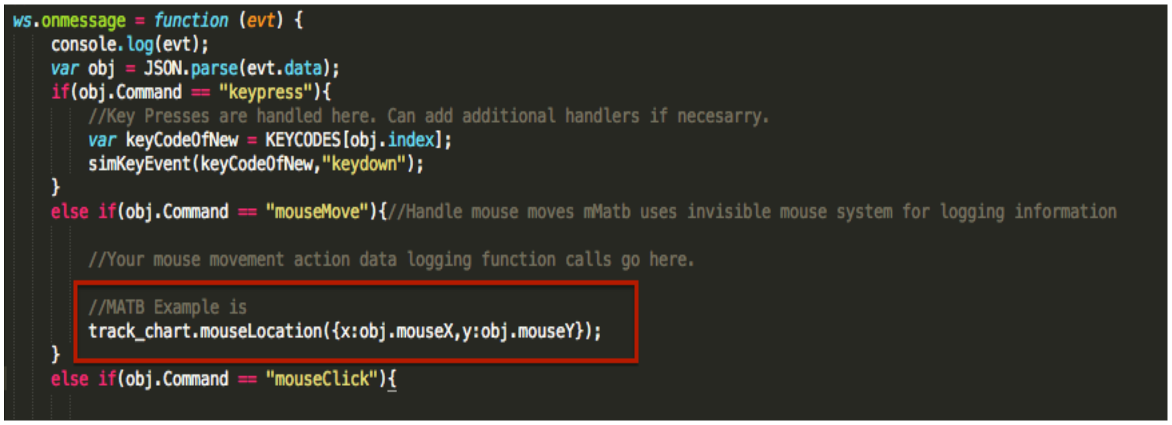


```
(set-task "actr.tasks.ActrInterface.SIMCogACTR")
```



```
(set-task "actr.tasks.ActrInterface.SIMCogACTR")
```

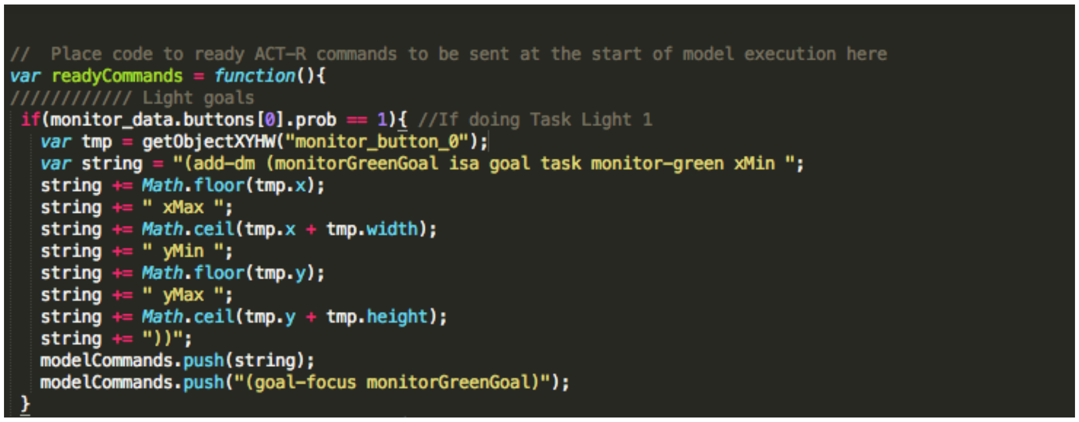
2. Recording Mouse Movement - To add recording of mouse movements in the mMATB task, a call is made to the same function being used to log data in the real task but the parameters are replaced with the ACT-R model data. This would look like:



```
ws.onmessage = function (evt) {
    console.log(evt);
    var obj = JSON.parse(evt.data);
    if(obj.Command == "keypress"){
        //Key Presses are handled here. Can add additional handlers if necesarry.
        var keyCodeOfNew = KEYCODES[obj.index];
        simKeyEvent(keyCodeOfNew,"keydown");
    }
    else if(obj.Command == "mouseMove"){//Handle mouse moves mMatb uses invisible mouse system for logging information

        //Your mouse movement action data logging function calls go here.

        //MATB Example is
        track_chart.mouseLocation({x:obj.mouseX,y:obj.mouseY});
    }
    else if(obj.Command == "mouseClick"){
```

It's important to note that the version of mMATB provided in the download does not record data, so this mouse location recording code is only provided as an example of how recording would be done.

3. Generating ACT-R Commands – Generating ACT-R commands for the mMATB is quite useful. The components in the task can be different sizes and be arranged differently depending on the window size. This can be problematic if the model does not learn the location of visual elements. This can easily be dealt with by generating an ACT-R command in the *readyCommands* function to provide the location of visual elements with declarative memory chunks or with the location information provided in goal chunks. An example of programmatically specifying a goal chunk containing the location of the green button is given below.



```
//  Place code to ready ACT-R commands to be sent at the start of model execution here
var readyCommands = function(){
//////////// Light goals
 if(monitor_data.buttons[0].prob == 1){ //If doing Task Light 1
   var tmp = getObjectXYHW("monitor_button_0");
   var string = "(add-dm (monitorGreenGoal isa goal task monitor-green xMin ";
   string += Math.floor(tmp.x);
   string += " xMax ";
   string += Math.ceil(tmp.x + tmp.width);
   string += " yMin ";
   string += Math.floor(tmp.y);
   string += " yMax ";
   string += Math.ceil(tmp.y + tmp.height);
   string += "))";
   modelCommands.push(string);
   modelCommands.push("(goal-focus monitorGreenGoal)");
 }
```

4. Using !bind! – Sometimes the modeler may want to evaluate custom statements within the model. For example, a modeler may want the model to be able to determine how many left-arrow key presses the model must perform in order to increase the frequency of a channel in the Communications task. Until a cognitively plausible set of productions is implemented to handle this task, a !bind! function can be used to determine this value within a single production. The following fabricated production utilizes a !bind! for such a purpose:

```
(p calculate-key-presses
   =goal>
      task communication
      target-frequency =target-freq
     current-frequency =current-freq
==>
   !bind! =keypresses (get-freq-diff =target-freq =current-freq)
   =goal>
      n-keypresses = keypresses
)
```

Then, within SIMCogACTR.java, one must implement the bind() method to handle the "get-freq-diff" statement. The following serves that purpose by calculating the difference between the first and second arguments, and then dividing the difference by two to account for the fact that each key press changes the frequency by 0.2:

```
public double bind(Iterator<String> it) {
     it.next(); // The first string is the "("
     String cmd = it.next(); // The second string is the command
     if(cmd.equals("get-freq-diff")) {
          String targetString = it.next(); // The third string is the first argument
          String currentString = it.next(); // The third string is the second argument
          int targetFreq =
               (int) (Double.parseDouble(targetString.replace("\"", "")) * 10);
```

```
        int currentFreq =
            (int) (Double.parseDouble(currentString.replace("\"", "")) * 10);
        return (targetFreq – currentFreq)/2;
    } else …
```

7. Rules for Specifying Screen Objects

To properly declare screen objects that will translate interface elements into ACT-R visual chunks, follow the instructions outlined in this section. At the top of the simCog.js file, there is an array called *screenObjects*. This is where interface elements that will be sent to ACT-R are declared. Each such interface element is a JSON object within this array. There can be as many screen objects in this array as desired.

There are 6 valid attributes (i.e., key-value pairs). All screen objects must have an ID and type attributes. All additional attributes add details about the objects in the model's representation or improve system use.

Any screen object may be assigned the attributes *change*, *color*, and/or *clickable*. The screen object types *OvalOutlineFill* and *RectangleOutlineFill* can be assigned an attribute called *secondary color*. The object types *Line*, *Label*, and *Rectangle* can have a *task relevance* attribute. Each of these attributes is discussed below.

7.1 Screen Object Attributes

7.11 ID

Key: id
Values: Any unique String or Object with DOM node and name.
Notes: Use Object when element has no id or that id isn't unique.

ID is used to get the DOM element that will act as a source for information. ID is declared as the unique string ID of the DOM interface element that this screen object will represent.

Not all DOM interface elements have unique identifiers. If the interface element does not have a unique ID, then the ID can be declared as an object with two attributes: *name* and *domLocation*. *Name* is a string that is unique to the specific screen object; this can be any string the experimenter wants as long as it is unique.[4] The value of *domLocation* is assigned the DOM tree node of the interface element this screen object will represent. The tree node can be found by traversing the DOM tree using parent/child selectors or using a relationship to another interface element that has a DOM ID (i.e., the parent of a DOM object with an ID).

[4] Note that for debugging purposes, and for sharing code between modelers and experimenters, it helps if object IDs are meaningful.

7.12 Type

Key: type

Values: “Line”, “Label”, “Cross”, “Rectangle”, “Oval”, “RectangleOutline”, “RectangleOutlineFill”, “OvalOutline”, or “OvalOutlineFill”

Type is the string identifier for one of the nine valid types*: Line, Label, Cross, Rectangle, Oval, Rectangle Outline, RectangleOutlineFill, OvalOutline, and OvalOutlineFill*.

Two examples of screen object declarations with only Type and ID specified are:

```
var screenObjects = [
        {type : "RectangleOutline",
                id : {
                        domLocation:document.getElementById
                                        ("#management_tank_0_resource").parentElement,
                        name:"management_tank_0_resource_tank_frame"
                }
        },

        {type : "RectangleOutlineFill", id : "resource_switch_0"}
]
```

7.13 Change

Key: change

Values: “static”, “poll”, [“poll”, “attr”,“attr”], [“attr”, “attr”]

Default: Event based.

Notes: See 7.2 for specifics on which attributes can change for each type

Change defines how the screen object will be updated for ACT-R. For performance reasons, *change* is a useful setting to define on systems with a very large number of screen objects. While there haven’t been any performance issues with mMATB (which contains roughly 80 interface elements), there are potential problems as the size and number of interface elements in a task increases. These potential problems would arise from network delays due a large amount of messages being sent or from computational demands inherent in performing calculations on numerous screen objects. This wouldn’t have a huge effect on the system and could potentially be mitigated with more computational resources.

There are three types of updating settings that the *change* attribute can be set as: *Polling*, *Event-Based*, and *Static*. Each setting has its own advantages for specific interface elements that improve overall performance.

Screen objects declared as *static* never update. This means that these objects will only be sent over in the *init* message. This setting is useful when an interface element is needed in the model but doesn't change throughout the task (e.g., a reference line or target area). To declare a screen object as static, add a change attribute to the screen object and set it to the string "static". An example of this declaration is:

{type : "Rectangle", id : "background_0", change: "static"}

Screen objects declared as *Event-Based* will update whenever a mutation observer detects a change within the DOM interface element. This is useful when interface elements change somewhat infrequently. An example might be an object that changes color every second. All screen objects will default to *Event-Based* whenever change isn't included in a declaration.

Screen objects declared as *Polling* will be updated on a fixed time interval. This time interval defaults to 60 times per second (approximately 16.666 ms); the user can modify this by changing one variable (*refreshRate*). Interface elements that will benefit performance by being declared as *Polling-Based* are constantly changing. This improves performance overall because they will not be spamming the system with messages frequently as they change small amounts. Additionallly, multiple update messages are aggregated into a single message. To declare an object as *Polling*, add the change attribute to the screen object and set it to "poll". This would look like

{type : "Rectangle", id : "moving_Circle_0", change: "poll"}

When an interface element is updated, all visual attributes are updated by default. To further improve performance, the user can change which attributes are sent when the object is updated. For example, if a rectangle only changes color but not size or position, then the location, height, and width do not need updating as they will stay the same throughout the task. While this may not seem a massive increase in performance, specifying updates selectively will eliminate a large amount of redundant information in the message passing throughout the system's execution.

To specify which attributes are updated in a screen object declaration, the change attribute is set to an array of updateable values. The list of updateable values is "x", "y", "height", "width", "color", "secondaryColor", and "stringVal". If an object is *Polling-Based*, then the first element in this array must be "poll". Several sample declarations of screen objects with specific updateable items are:

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

{type : "Rectangle", id : "comm_target_rect", change : ["poll" , "color" , "y"]}

{type : "OvalOutlineFill", id : "track_circle_0", change :["x", "y", "height", "width"]}

7.14 Color

Key: color

Values:

"CSS color",

[{class: “classname”, real: “CSS color”},{class: “classname”, real: “CSS color}],
[{style: “stylename”, real: “CSS color”},{style: “stylename”, real: “CSS color}]

Default: Automatic extraction using jQuery dependent DOM computations
Notes: Automatic extraction is easiest, use a manual definition when specific color configurations are desired.

Color defines the color or color scheme of an object. This is automatically extracted for all objects by default using CSS specificity computations. These computations are reliant on jQuery to execute. If there is a need to manually set color, it can be done as follows:

In the simple case where the interface element is one color that will never change, color is declared as a string of any valid CSS color (See Appendix C for a full list). An example of the simple case is:

```
{type : "Rectangle", id : "moving_rectangle_1", color: “yellow green”}
```

If an object can have multiple colors changes, then you will need to use a more advanced solution. This solution sets color to an array of key-value pairs. Each key is the CSS class or inline style name that the interface element has. Keys relating to CSS classes must be declared as “class”; keys relating to inline styles must be “style”. Each value is a valid CSS color. For example, if a rectangle were to have the class “highlighted” when it is red and “normal” when it is green, then the color declaration would look like:

```
{id: “changingRectangle”, type: “Rectangle”,
        color : [{class: “highlighted”, real: “red”},{class: “normal”, real: “green”}]
}
```

7.15 Secondary Color

Key: secondaryColor
Values:
“CSS color”,
[{class: “classname”, real: “CSS color”},{class: “classname”, real: “CSS color}],
[{style: “stylename”, real: “CSS color”},{style: “stylename”, real: “CSS color}]

Default: Automatic extraction using jQuery dependent DOM comuptations
Notes: Only OvalOutlineFill and RectangleOutlineFill object types may have this attribute.

Secondary color refers to the border of an object and is therefore only used in object types with two colors (*OvalOutlineFill*, *RectangleOutlineFill*). Secondary color is extracted automatically in the same way as color is. Secondary color can be manually specified in the same way as color.

A manual secondary color *screenObject* might look like

```
{type : "RectangleOutlineFill", id : "comm_channel_0_rect",

        secondaryColor : [{style:"fill: #90ee90;",real:"green"},{style:"fill: #ff0000;",real:"red"}],

        color : "black"

}
```

7.16 Clickable

Key: clickable

Values: true, false

Default: false

Notes: No reason to add this to an element if it's not clickable; default handles it.

Clickable assists in the handling of click events and can be added to any object. Clickable objects are objects that the server recognizes as relevant to the client for clicking. The reasoning for this feature is due to the asynchronous nature of the system. While delays transmitting messages average roughly 1-2 milliseconds (when both the server and client are running locally), this may still be too slow for some fast moving interface elements. Clickable objects fix this. When the server registers a click by the model, it will check if the click was within any of the clickable objects and send the IDs of those it clicked client side with the click message. This allows the client to generate the click on that item regardless if the object has moved several pixels and doesn't contain the click location anymore. To declare an object as clickable, simply add a *clickable* attribute and set it to *true*. This would look like:

```
{id: “changingRectangle”, type: “Rectangle”, clickable:true}
```

7.17 Task Relevance

Key: taskRelevant

Values: true, false

Default: true

Notes: Only line, label, and rectangle object types may have false values for taskRelevant. Objects with a false task relevance attribute won't be in the ACT-R visicon.

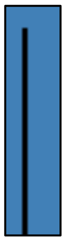
Task relevance refers to an object being visible to the model. Task-irrelevant items are useful for the modeler to help better visualize things (e.g.,, a background to better view objects). All screen objects default to task relevant. The only types of screen objects that can be declared as task irrelevant are *Line, Label,* and *Rectangle*. To declare a screen object as task irrelevant, add a *taskRelevant* attribute and set it to *false*. This might look like:

{type : "Rectangle", id “background_0” :, taskRelevant : false}

7.2 Example Specifications

Below are examples of every item available for usage by the system. Each example contains an image of the shape, a sample screen object declaration, a list of what is must be extracted for the server, list of optional items, and a sample JSON object of what might be sent to the server.

Example 1. Line – Simple line, cannot be diagonal currently.



Server Required: ID/X/Y/Height/Width/Type

Optional: Color, Task Relevance, and Clickable.

Sample screenObject declaration:

{type : "Line", id : "UID", change : "static", color : "Black"}

Sample server JSON:

{id: " UID ", x : 10 , y: 10, height: 10, width: 10, type: "Line", color:" Black "}

Example 2. Label – Simple Textbox, no visible box surrounding, just text

Server Required: ID/X/Y/Height/Width/Type

Optional: Color, Task Relevance, and Clickable.

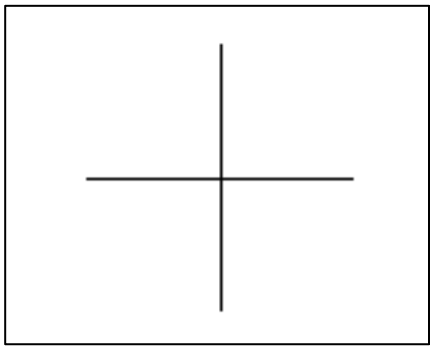
Sample screenObject declaration:

{type : "Label", id : " itemID ", change : ["evt","color","stringVal"], color : [{class : "a", real: "red"}, {class : "b", real: " black "}]}

Sample server JSON:

{id: "itemID", width: 22, height : 23, x: 24, y: 25, stringVal : “text”, type : "Label", color : “black”}

Example 3. Cross – Simple cross, useful for crosshairs.



Server Required: ID/X/Y/Height/Width/Type

Optional: Color, and Clickable.

Sample screenObject declaration:

{type : "Cross", id : "UID", change : "static", color : " black "}

Sample server JSON:

{id : " UID ", width : 22, height : 22, x:22, y: 22, type : "Cross", color : “black”}

Example 4. Rectangle – Creates a simple Rectangle, it is filled with the provided color.

Server Required: ID/X/Y/Height/Width/Type

Optional: Color, Clickable, and Task Relevance

Sample screenObject declaration:

{type : "Rectangle", id : "itemID", change: “color”, color : "green"}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:25,color:"green",type:"Rectangle",height:33 }

Example 5. Oval – Creates a simple oval, it is filled with the provided color.

Server Required: ID/X/Y/Height/Width/Type

Optional: Color, Clickable.

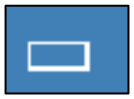
Sample screenObject declaration:

{type : "Oval", id : “itemID”, change : "static", color : "black"}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:25,color:"alice blue",type:"Oval",height:33 }

Example 6. RectangleOutline – Creates a rectangle frame. This is useful for a container for a resource or other object.



Server Required: ID/X/Y/Height/Width/Type/Thickness (Wall thickness)

Optional: Color, Clickable

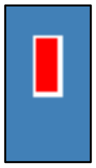
Sample screenObject declaration:

{type : "RectangleOutline", id : "itemID", change : “static”, color : "white"}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:25,color:" white",type:"RectangleOutline ",height:33, thickness:3 }

Example 7. RectangleOutlineFill – Creates a rectangle with a 2-color system. Color, the first color is the color of the fill. Secondary color is the color of the border



Server Required: ID/X/Y/Height/Width/Type/Thickness (Wall thickness)

Optional: Color, Secondary Color, and Clickable

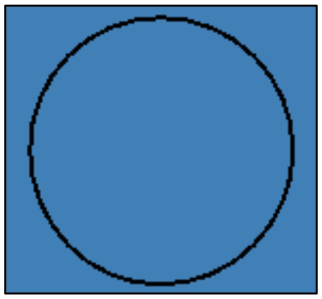
Sample screenObject declaration:

{type : "RectangleOutlineFill", id : "itemID", color : "red", secondaryColor : “white”}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:25,color:"red",secondaryColor: “white”, type:" RectangleOutlineFill ",height:33, thickness:3 }

Example 8. OvalOutline – Creates an Oval with only an outline stroke.



Server Required: ID/X/Y/Height/Width/Type/Thickness (Wall thickness)

Optional: Color, Clickable

Sample screenObject declaration:

{type : "RectangleOutline", id : "itemID", change : “static”, color : "black"}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:25,color:" black ",type:"OvalOutline",height:33, thickness:3 }

Example 9. OvalOutlineFill – Creates an Oval with a 2-color system. Color, the first color is the color of the fill. Secondary color is the color of the border

Server Required: ID/X/Y/Height/Width/Type/Thickness

Optional: Color, Secondary Color, and Clickable

Sample screenObject declaration:

{type : "RectangleOutlineFill", id : "itemID", color : "green",secondaryColor : “white”}

Sample server JSON:

{id : "itemID”, x:99, y: 88, width:33 ,color:"green", secondaryColor: “white”, type:" OvalOutlineFill", height : 33, thickness : 3 }

8. Data Logging Capabilities

SIMCog-js allows for the logging of detailed model data. This is useful for analyzing model errors and decisions that occur. To turn on data logging edit the SIMCog.js file and set the *recordActrLogFile* (found in the Debugging and Data Recording Settings, under screen object declarations) to *true*. This will generate a JSON and CSV file containing the following initial information:

- Initial UNIX time
- Task Name

- Loaded production rules

The system will also generate data on the model's task execution. This isn't updated on a regular interval due to how events occur in ACT-R. Every time a change is detected the following is recorded:

- Time since start of task
- Declarative memory contents
- Visual Objects
- Model eye location
- Events
- Fatal Errors
- ACT-R buffer contents

This file will be named with a time stamp and can be found in the logs folder.

It is important to note that to not all information for analysis may be readily available for usage. For example, it may take some brief analysis to determine when a rectangle changed color. This issue could be mitigated by using the task data (client side) in conjunction with the output file to better determine points of interest.

9. Future Work

Planned future functionality for SIMCog-JS includes:

- Implement the server side into other models like EPIC and Python mathematical models.
- Try to get both systems running on the same internal clock in order to allow for faster-than-real-time modeling and prediction.
- Add ability to support sound based stimuli for the model.

Appendices

Appendix A. Browser Support

Working (Tested on, assumed to work on releases between)

Safari Version 7.0.6. Released August 2014~

Google Chrome Version 31.0.1650.57. Released November 2013

Google Chrome Version 39.0.2171.95. Released November 2014 (See differences)

Firefox 33.0.3. Released November 10th 2014

Firefox 34.0.5. Released December 1st 2014

Known Browser differences

1. The way inline styles are represented within the DOM with newer versions of Chrome has changed, so manual color definitions within screenObjects array must change accordingly if you are switching between browser versions. Previous browser versions use "fill: #90ee90;" newer versions use "fill: rgb(144, 238, 144);". Firefox stays consistent with the "fill" definition throughout all recent versions.

Appendix B. ACT-R Key codes

{"escape", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "f8", "f9", "f10", "f11", "f12", "f13", "f14", "f15", "", "", "", "", "", "", ""},
{"backquote", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-", "hyphen", "=", "delete", "help", "home", "pageup", "clear", "=", "/", "*", ""},
{"tab", "q", "w", "e", "r", "t", "y", "u", "i", "o", "p", "[", "]", "backslash", "", "forward-delete", "end", "page-up", "", "keypad-7", "keypad-8", "keypad-9", "keypad-hyphen"},
{"caps-lock", "a", "s", "d", "f", "g", "h", "j", "k", "l", "semicolon", "quote", "return", "return", "", "", "", "", "", "keypad-4", "keypad-5", "keypad-6", "keypad-plus"},
{"shift", "z", "x", "c", "v", "b", "n", "m", "comma", "period", "dot", "/", "right-shift", "right-shift", "", "", "up-arrow", "", "", "keypad-1", "keypad-2", "keypad-3", "keypad-enter"},
{"left-control", "left-option", "left-command", "spc", "spc", "spc", "spc", "spc", "spc", "spc", "spc", "right-command", "right-option", "right-control", "", "left-arrow", "down-arrow", "right-arrow", "", "keypad-0", "keypad-0", "keypad-period", "keypad-enter"}

Appendix C. VALID CSS COLORS

To see each visit http://www.w3.org/TR/SVG/types.html#ColorKeywords

BLACK	BLUE	CYAN	DARK GRAY	GRAY
GREEN	LIGHT GRAY	MAGENTA	ORANGE	PINK
RED	WHITE	YELLOW	ALICE BLUE	ANTIQUE WHITE
AQUA	AQUA MARINE	AZURE	BEIGE	BISQUE
BLANCHED ALMOND	BLUE VIOLET	BROWN	BURLEY WOOD	CADET BLUE
CHARTREUSE	CHOCOLATE	CORAL	CORNFLOWER BLUE	CORNSILK
CRIMSON	DARK BLUE	DARK CYAN	DARK GOLDEN Rod	DARK GREEN
DARK KHAKI	DARK MAGENTA	DARK OLIVE GREEN	DARK ORANGE	DARK ORCHID
DARK RED	DARK SALMON	DARK SEA GREEN	DARK SLATE BLUE	DARK SLATE GRAY
DARK TURQUOISE	DARK VIOLET	DEEP PINK	DEEP SKY BLUE	DIM GRAY
DODGER BLUE	FIRE BRICK	FLORAL WHITE	FOREST GREEN	FUCHSIA
GAINSBORO	GHOST WHITE	GOLD	GOLDEN ROD	GREEN YELLOW
HONEY DEW	HOT PINK	INDIAN RED	INDIGO	IVORY
KHAKI	LAVENDER	LAVENDER BLUSH	LAWN GREEN	LEMON CHIFFON
LIGHT BLUE	LIGHT CORAL	LIGHT CYAN	LIGHT GOLDEN	LIGHT GREEN
LIGHT PINK	LIGHT SALMON	LIGHT SEA GREEN	LIGHT SKY BLUE	LIGHT SLATE GRAY
LIGHT STEEL BLUE	LIGHT YELLOW	LIME	LIME GREEN	LINEN
MAROON	MEDIUM AQUA MARINE	MEDIUM BLUE	MEDIUM ORCHID	MEDIUM PURPLE
MEDIUM SEA GREEN	MEDIUM SLATE BLUE	MEDIUM SPRING GREEN	MEDIUM TURQUOISE	MEDIUM VIOLET RED
MIDNIGHT BLUE	MINT CREAM	MISTY ROSE	MOCCASIN	NAVAJO WHITE
NAVY	OLD LACE	OLIVE	OLIVE DRAB	ORANGE RED
ORCHID	PALE GOLDEN ROD	PALE GREEN	PALE TURQUOISE	PALE VIOLET RED
PAPAYA WHIP	PEACH PUFF	PERU	PLUM	POWDER BLUE
PURPLE	ROSY BROWN	ROYAL BLUE	SADDLE BROWN	SALMON
SANDY BROWN	SEA GREEN	SEA SHELL	SIENNA	SILVER
SKY BLUE	SLATE BLUE	SLATE GRAY	SNOW	SPRING GREEN
STEEL BLUE	TAN	TEAL	THISTLE	TOMATO
TURQUOISE	VIOLET	WHEAT	WHITE SMOKE	YELLOW GREEN

Acknowledgments
SIMCog-JS was supported by the HUMAN Lab at AFRL and the Air Force Office of Scientific Research (AFOSR). Contracting support was provided by the Oak Ridge Associate Universities (ORAU).