Modeling Visual Search in Interactive Graphic Interfaces: Adding Visual Pattern Matching Algorithms to ACT-R

Farnaz Tehranchi (farnaz.tehranchi@psu.edu)
Department of Computer Science and Engineering
Penn State, University Park, PA 16802 USA

Frank E. Ritter (frank.ritter@psu.edu)
College of Information Sciences and Technology
Penn State, University Park, PA 16802 USA

Abstract

We provide an update on JSegMan, an interactive system to extend the ACT-R cognitive architecture to interact with dynamic interfaces based on the screen contents and generating input for the operating system directly. Current ACT-R models typically interact with the world through ACT-R's device interface—an abstract representation of the world that is based on a simulated Lisp environment provided with ACT-R, or by instrumenting interfaces. In JSegMan, computer vision pattern matching algorithms and visual patterns extend the ACT-R cognitive architecture. With JSegMan, models directly move the cursor on the screen, click on application GUI objects on PCs, and type through the use of existing Java libraries. Implementing users' visual search strategies and input abilities for different visual objects enables the detailed modeling of interactive tasks on any interface. The visual pattern matching algorithms serve two goals: to simulate user behavior in interactive tasks and to create representations of visual stimuli. We tested our visual pattern matching approach by using it with an existing model for a long spreadsheet task. We found that the revised model more accurately predicted a 20-min task by entirely performing the task on an uninstrumented and unmodified interface.

Keywords: Cognitive model; Cognitive architecture; Human-computer interface, interaction; Perception and motor output; Computer vision; Simulated eyes and hands.

Introduction

Cognitive architectures are programming languages specifically designed for modeling unified theory of all human cognition, such as Soar (Laird, 2012; Newell, 1990) and ACT-R (Anderson, 2007). Using models as users have been envisioned before (e.g., Byrne, 1994; Lohse, 1997), but has not yet been widely applied. These models require simulating an interactive behavior.

ACT-R's interactive behaviors include moving the cursor, clicking, and typing. Although cognitive architectures such as ACT-R provide powerful and flexible frameworks for modeling general human behavior, only a portion of their capabilities are often used to capture many complex real-world behaviors (Fu et al., 2003), because the skills can only be applied with unmodified ACT-R in specialized windows provided with ACT-R. Additionally, cognitive models need to interact with interfaces to be useful in human-computer interaction (HCI) (Byrne & Kirlik, 2005; Kieras, 2009; Ritter et al., 2000).

A Cognitive Model Interface Management System (CMIMS), which is an extension of the User Interface Management System (UIMS) concept (Ritter et al., 2001) is an approach to provide models access to interfaces so that cognitive models can be applied more routinely in HCI.

ACT-R/PM (Byrne & Anderson, 1998) is a CMIMS that allows ACT-R models to interact with interfaces built within a special Common Lisp window. Another successful version of the interactive model is Salvucci's (2006) driver model. Salvucci also introduced an ACT-R system with an implementation of the ACT-R cognitive architecture in the Java programming language. It can be used as a library in other ACT-R projects (Salvucci, 2009, 2013).

In all these approaches, it will be difficult to reuse these models (their task knowledge) because of the nature of the embedded task environment—the models can only interact with their instrumented interfaces. The interfaces that have been modified to provide the models access to information on the display in that application. Also, the interfaces have been adjusted to accept commands directly from the models. Even in JSON ACT-R, which makes a general approach available for these modifications (Hope, Schoelles, & Gray, 2014), reconfiguring the connection is challenging.

We take inspiration from SegMan, segmenting the screen to different features (e.g., based on color) and manipulating interface usage. SegMan was productive and was used to interact with a wide range of interfaces (Ritter, Kukreja, & St. Amant, 2007; St. Amant et al., 2005; St. Amant & Riedl, 2001). However, SegMan is hard to use and maintain at this point. We have also explored providing a connection to any task in Emacs, ESegMan, such as pressing a key, moving a cursor, clicking a mouse, and moving attention (Tehranchi & Ritter, 2017). ESegMan could only manipulate Emacs window and is limited to Emacs.

These results have led us to design JSegMan, a system written in Java that extends ACT-R to provide models with interaction with any interface on a PC, and thus the world.

We report elsewhere its hands model that provides typing and mouse interaction (Tehranchi & Ritter, 2018). JSegMan creates a way to interact with all interfaces using an extended Java library to input motor commands (keystrokes, mouse moves, and mouse clicks).

We report here the first vision system for cognitive models to see objects on the screen using pattern recognition and other computer vision algorithms.

As an example application, we use ACT-R and JSegMan to perform the Dismal spreadsheet task using an existing large (500 rule) ACT-R model (Paik, Kim, Ritter, & Reitter, 2015). The revised model with JSegMan predicted the response time more accurately while, importantly, using the same but unmodified interface that the human subjects used. Finally, we provide conclusions, lessons, and insights.

Visual Attention and Environment

There are theoretical and empirical challenges in understanding visual attention and implementing its mechanism. Constructing a system to simulate attention based on eye movement data such as fixation and saccade can enhance our knowledge of perceptual-motor aspects of the human system. Besides cognition, perception, and action are also essential for resembling the models' behavior closer to the human behavior. The focus of cognitive analysis is often on cognition and response time, rather than an analysis of interaction and knowledge related to the interaction. Given the visual nature of most current user interfaces, the vision module is relatively central in modeling most HCI tasks. The vision module is used to determine what ACT-R/PM "sees" (Byrne, 2001).

Each object on display will be represented by one or more features in the vision module's icon. The vision module creates chunks from these features that provide declarative memory representations of the visual scene. These chunks are visual location and visual object types. Production rules' constraints can match chunks. After the vision module creates a chunk representing an object, visual attention must be directed to the location of that object, through a visual location chunk.

Anderson et al. (1998) introduced the visual hunt-feature and found-target in which there are the equivalents of move attention and move the cursor in current ACT-R. The vision module move-attention operator shifts attention to a location; ACT-R/PM must have representation for those visual locations.

Salvucci (2001) distinguished eye movements and shifts of attention. The ACT-R visual mechanism mainly considers moving attention to be the same as a saccade and reflects a top-down search without backtracking. Furthermore, the visual mechanism skips items that do not have the same features as the target features. Kieras and Meyer (1997) in EPIC predicted that eye movement patterns at least in their task, menu search, are 50% top to bottom and 50% random (Hornof & Kieras, 1997). The EPIC inspired visual mechanism looks for the target object under the current location rather than starts at the top of the display; current ACT-R does not implement this mechanism.

Static Environment: In this kind of environment, the interface displays the application semantics with which the user interacts directly and immediately. No other processes can modify the environment and there will be only one level of commands. For example, we have gathered data by showing users pictures in a PDF document and having them click on a target object. The object does not change when clicked upon.

Interactive Environment: In this kind of environment, the interface content only changes when the user provides input. For instance, in Emacs, the display typically only changes based upon the user input.

Dynamic Environment: These user interfaces often have a high degree of interaction and are complex, and may be non-deterministic to the user. The boundary between application and user interface is difficult or impossible to control, and what the user can see may change without the user's input. The screen content changes based on both user input and time. Therefore, it becomes problematic to decide if this interaction should be handled by the model or in the application level (Soegaard, 2013).

The JSegMan system is intended to allow modelers to participate more directly in all of these environments, particularly the interactive and dynamic environments to study computer behavior.

In this paper, JSegMan has been examined in the Dismal spreadsheet mode of Emacs, which requires a nested level of commands (Ritter & Wood, 2005). Dismal is a spreadsheet developed for Emacs that was designed to gather and analyze behavioral data. Users can interact with the spreadsheet similar to other Emacs windows and also with Dismal function calls. Dismal is an interactive environment that its interface components change by user input and commands.

Visual patterns in JSegMan are small images, a representation of an object in the visual scene. Figure 1 shows different patterns on the screen. Visual patterns in JSegMan are equivalent to visual location and visual object chunks in ACT-R. ACT-R can access all required objects of the current visual scene by defining the patterns as independent patterns or a logical mix of patterns. JSegMan needs to use multi-part patterns to interact with more complicated environments. Modelers create required visual patterns for JSegMan.

In the Dismal model, to find a spreadsheet cell, as shown in Figure 1, the user first finds the column and then looks for the row. To implement this action in JSegMan, we have to find the pattern, re-set the overlap region, and find the new pattern (e.g., the row) inside a pattern. This nested pattern finding will make the models more human-like by requiring additional knowledge and steps that take additional time and allow the opportunity for additional errors. To add this functionality, the matching algorithm needs some higher level of built-in logic.

In JSegMan, the target visual pattern is compared with all the same size available frames in the screen. It moves the comparison frame one pixel at a time, a constant distance in each saccade, from left to right and top to bottom.

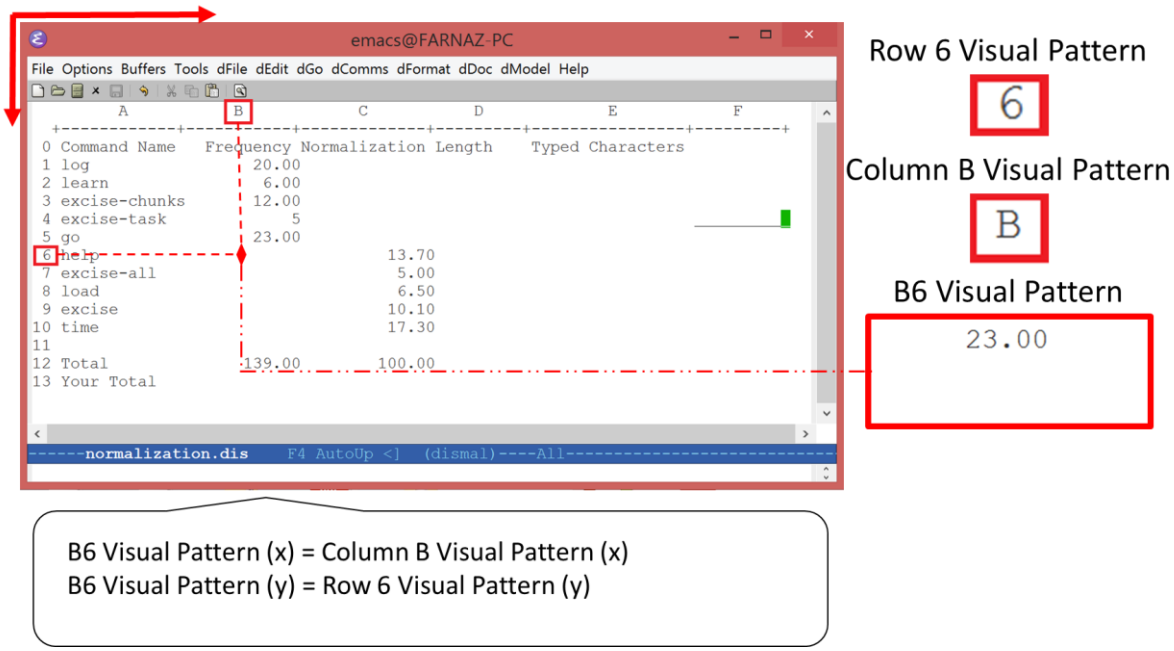

Figure 1. Defining visual patterns in an interactive environment.

This approach uses patterns instead of regular visual objects. Therefore, visual object features in ACT-R such as size and location are not practical for the model. Consequently, any changes in the visual scene cannot affect patterns such as changes in screen resolution and size. The model can still find the target visual object because visual attention does not change by screen-x and screen-y. Instead, the model finds/re-finds the pattern in the visual scene. From Java process, JSegMan can use these features and update the visual representation in ACT-R. This approach is independent of the task environment's system because the location of task environment on the screen does not change the underlying patterns.

To support working with interaction interfaces, we propose two methods: (a) The model always has a knowledge of the dynamic environment because the model has to respond appropriately to manipulation of said environment (Anderson, Farrell, & Sauers, 1984; Soloway & Johnson, 1984), and (b) The model uses pattern matching logic. The next section explains these methods in details.

Visual Pattern Matching Algorithms

We used two Java packages: (a) Robot and (b) Sikuli (Yeh, Chang, & Miller, 2009) to develop the JSegMan system. JSegMan functions are divided into two primary sections. The first, hand simulation, includes motor module methods, *move cursor to*, *get mouse coordinates*, *handle click*, *handle keypress*, and *type word*. The second section, the eye simulation, includes *move attention*, *process display*, and *print visicon* (a debugging command).

Java Robot is used to automate operating system actions such as clicks and typing. The Robot package implements actions from the ACT-R motor module, such as moving the mouse as moving the attention pointer by ACT-R, clicking a mouse, and pressing keystrokes.

The experimental environment is an application GUI that contains the information of visible objects such as labels, text fields, images, buttons, links, radio buttons, and toggle buttons. With Sikuli, we have access to all these objects through the screen bitmap and can define them as Java objects (Kasper, Correll, & Yeh, 2014; Yeh, Chang, & Miller, 2009).

With Sikuli, cognitive models can identify and control GUI components, anything the end users could interact with and see on the screen (not just from Java applications), and also could support reading from text recognition (OCR). More specifically, it uses GUI screenshots for searching patterns to direct mouse and keyboard events in contrast with seeking screen locations, which may change during the experiment. Also, it utilizes Template Matching, a pattern-matching algorithm in OpenCV, an open-source computer vision library in which a pattern (small image) is compared against the overlapped image regions (the computer screen). Algorithm 1 summarizes the methods. Both the display and the visual pattern pixels matrix are the input of matchTemplate (Bradski, 2004). In the end, it returns the location with a higher matching value. This algorithm has been implemented in Sikuli.

Algorithm 1: Visual Pattern Matching	
1	Input: Visual Pattern pixels matrix Display pixels matrix Output: Result matrix in the size of the display, maximum value of the Result matrix
2	matchTemplate(Display, Visual Pattern, Result, CV_TM_CCOEFF_NORMED);
3	Return MaxLocation(Result);

First, JSegMan loads the visual patterns by processing the display. Then, the model uses pattern-matching algorithms

to find the pattern on the screen and then move the visual attention. The current visual scene is a screenshot of a computer screen.

Patterns should be distinct enough for JSegMan to be identified uniquely. The pattern-matching algorithm is color sensitive. There are some suggested approaches to use grayscale (e.g., Kuchhal, 2014). When JSegMan is looking for a visual pattern on display, one or more locations may be matched by the pattern matching process, depending on the number of objects on the display, display complexity, and the production rules' constraints.

The JSegMan process will check the pattern existence. When the JSegMan process is running, the ACT-R process pauses and resumes when JSegMan finishes the eye and hand simulation. Therefore, the JSegMan task completion times will not affect the ACT-R theoretical response time, but the system together does run modestly slower in real time. Currently, JSegMan takes 195.5 ms for pressing a keystroke (Burns, Ritter, & Zhang, 2016) and the eyes simulation delay takes 500 ms to do search tasks; JSegMan searches up to 1000 ms to find patterns in real time. If the JSegMan process cannot find the pattern, it will return an error message and pause the ACT-R process.

Furthermore, ACT-R passes a Value slot of the vision object, the name of the pattern, to the Java process. Finally, the task environment responds successfully to each of the requests made by the ACT-R model, and the model is able to create and attend to objects within the dynamically/interactively changing visual scene.

Figure 1 distinguishes logic pattern matching and regular matching. For example, to shift attention to cell B6 in the spreadsheet, we can either define a unique independent pattern for B6, the *B6 visual pattern* or define two patterns: one for its column and one for the row. Then, JSegMan calculates the exact location of B6 and move there directly. By collecting not only the reaction time data but also the eye-tracking data, we can predict how humans find a cell in the spreadsheet environment, and image recognition will have a more natural approach. In the Dismal model, the cell's B7 visual pattern is dependent on the B6's cell value. The Sikuli screenshot search engine can match the B7 visual pattern after the change in B6 is affected because the B7 visual pattern contains the B6's cell value. Using a logic pattern matching methods eliminates this dependency.

The Sikuli script operates only in the visible screen space and does not work on invisible GUI elements, such as those hidden beneath other windows, in another tab, or scrolled out of view. For instance, moving the cursor can cause a pop-up description which will block some patterns. Additionally, any UI-specific interaction, such as clicking the sidebar to scroll down the page, can be implemented by Sikuli classes. Further details, installation documents and instructions, and example models can be found on the project's website[1].

Revising the Dismal Model

To demonstrate the application of JSegMan, we tested our system with the Dismal model (Paik et al., 2015). In this study, data from 30 participants were collected while completing 14 subtasks of the Dismal spreadsheet task for mouse users. The Dismal spreadsheet subtasks were related to declarative and prosedueralize knowledge. The models' performance and participants' performance comparison was not completely realistic because the detailed trace of hand, fingers, and mouse movements were not modeled in much detail.

In our first attempt using JSegMan hands simulation, several missing keystrokes in the Paik et al.'s (2015) Dismal model had to be added, and the hands and fingers position on the virtual keyboard had to be adjusted. We were able to implement all the motor actions related to the keyboard in the original model. By redefining some of the keystrokes, we made more realistic key press actions in the virtual keyboard in ACT-R. For some keys that were not reachable by either the left or right hands, there had to be a request to the motor module to adjust the hand position. We added these requests to the original model.

In our analysis of the output of the model the keystrokes and mouse moves, we found missing actions such as clicking, including of which increased the total task time when compared to previous reports.

The Dismal spreadsheet environment is an interactive environment that changes based on user input. For instance, the *B6 visual pattern* in Figure 1 contains B5 cell's value. JSegMan simulates Dismal model eyes and hands correctly with this modification. The structure of the original Dismal model is based on pre-knowledge. Therefore, all individual patterns should be defined in advance without using the logic pattern matching strategy. The declarative chunks have been defined to move attention directly to cells rather than rows and columns. All patterns of cells are similar to the *B6 visual pattern* in Figure 1. We adjusted 162 declarative chunks in the original Dismal Model by adding a new slot for visual objects. In addition, to model eye movements, we added 52 new visual objects and visual locations. When JSegMan run with the Dismal model, we noticed a few missing sub-tasks (because the model did not produce a complete solution in the spreadsheet), so three new declarative chunks were added to the original model. The sequence of the subtasks for the visual module is not correctly interpreted. Therefore, we added a new constraint for the production rule that requests a new visual location be placed in the visual-location buffer.

Table 1 shows how the response time has been affected by our modification while the model learns over four trials. Besides proving the functionality of the JSegMan Dismal model, we were able to fit the Dismal model slightly better to the human data. The correlation shifts from 0.997 to 0.998 and reduced the mean square error (MSE).

[1] https://sites.psu.edu/ftehranchi/projects/

Table 1: The mean task completion time in seconds for the four learning sessions for the Dismal mouse-interface task (Paik et al., 2015) and correlation with the human data (N=30).

Day	Human		Original Model		JSegMan Correction Hands		JSegMan Correction Hands and Eyes	
	M	SE	M	SE	M	SE	M	SE
1	1366	60.8	1326	12.078	1338	12.06	1339	11.72
2	894	26.6	891	6.175	893	5.144	894	6.498
3	727	25.5	693	4.496	700	6.207	704	5.019
4	659	22.7	594	5.775	603	4.35	614	4.381
Correlation			0.997		0.9978		0.9984	
MSE			1747.5		1162.5		820.75	

Discussion and Conclusion

In this paper, we described our efforts to bridge the gap of interaction between cognitive models and task environments. This article focused on models written with the ACT-R cognitive architecture, but other architectures could use this approach and system.

We call our approach JSegMan because simulation eyes and hands in ACT-R models require the original ACT-R model, the Robot and Sikuli packages in Java, and currently Emacs (as a connector). The JSegMan approach will increase the usability, applicability, and accessibility of cognitive architectures. Also, it can be used as a cognitive model examiner—to see if the knowledge can do the task.

Using the JSegMan eyes and hands simulation in place of a user, questions about user interface designs such as evaluating designs, changing the interface and examining the effects on task performance can be answered more efficiently. The proposed approach can prove the advantages of CMIMS in HCI, and realize the use of user models in system design (Pew & Mavor, 2007). JSegMan can help implement this approach by testing user interfaces, making this process more approachable and more practical. Our results running the Dismal model and finding missing knowledge illustrates that JSegMan also offers the ability to understand a model more accurately.

The advantages of this approach are: (a) ACT-R code does not change due to the JSegMan, (b) JSegMan does not affect the ACT-R response time because JSegMan has separate timing formula and runs along with ACT-R, and (c) models are able to interact with any interface on a PC.

Further Research and Limitations

Further work remains. We still need to manually check if the action takes place correctly in JSegMan and the eyes follow the hands successfully, as well as exploring error generation and correction. Furthermore, we plan to use an object recognition algorithm to extract the visible objects without pre-defining them for models, but this functionality is beyond current ACT-R's action execution.

JSegMan should use the OCR capability more directly as it is likely more efficient and more comfortable to use. Additionally, the nested pattern matching that follows the EPIC visual search will be useful to implement. The current screen scanning approach is a bit mechanistic. The scan starts in the upper left every time. It is probably more realistic to start either where the previous task left off or based on other heuristics that people use (Hornof & Kieras, 1997).

In the future, we plan on offering an installation method that includes bundled versions of all dependencies, allowing near plug and play support with ACT-R. JSegMan components could also be expanded so JSegMan can observe the users, collect more realistic inputs, and thus better predict human performance. Therefore, JSegMan can be a substitute for humans in the software testing process and can be considered as a software testing tool.

Acknowledgments

This work was funded partially by ONR (N00014-15-1-2275). David Reitter provided useful comments on Emacs and Aquamacs (the Emacs version for Mac). We wish to thank Jong Kim who provided the idea for ESegMan and Dan Bothell for his assistance with ACT-R.

References

Anderson, J. R. (2007). *How can the human mind exist in the physical universe?* New York, NY: Oxford University Press.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8*, 87-129.

Anderson, J. R., Matessa, M., & Lebiere, C. (1998). The visual interface. In J. R. Anderson & C. Lebiere (Eds.), *The atomic components of thought*. Mahwah, NJ: Erlbaum.

Bradski, G. B. (2004). *Open source computer vision library*: Springer.

Burns, M., Ritter, F. E., & Zhang, X. (2016). Using Naturalistic Typing to Update Architecture Typing Constants. In *Proceedings of ICCM - 2016-14th International Conference on Cognitive Modeling*. University Park, PA: Penn State.

Byrne, M. D. (2001). ACT-R/PM and menu selection: Applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies, 55*(1), 41-84.

Byrne, M. D., & Anderson, J. R. (1998). Perception and action. In J. R. Anderson & C. Lebiere (Eds.), *The atomic components of thought*. Mahwah, NJ: Erlbaum.

Byrne, M. D., & Kirlik, A. (2005). Using computational cognitive modeling to diagnose possible sources of aviation error. *International Journal of Aviation Psychology, 15*(2), 135-155.

Byrne, M. D., Wood, S. D., Sukaviriya, P., Foley, J. D., & Kieras, D. E. (1994). Automating interface evaluation. In *Proceedings of the CHI'94 Conference on Human Factors in Computer Systems*, 232-237. ACM: New York, NY.

Fu, D., Houlette, R., Jensen, R., Bascara, O., & San Mateo, C. (2003). A visual, object-oriented approach to simulation behavior authoring. In *Proceedings of the Industry/Interservice, Training, Simulation & Education Conference (I/ITSEC 2003)*.

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*(4), 1007-1012.

Hornof, A. J., & Kieras, D. E. (1997). Cognitive modeling reveals menu search is both random and systematic. In *Proceedings of the CHI'97 Conference on Human Factors in Computer Systems*, 107-114. ACM: New York, NY.

Kasper, M., Correll, N., & Yeh, T. (2014). Abstracting perception and manipulation in end-user robot programming using Sikuli. In *Technologies for Practical Robot Applications (TePRA), 2014 IEEE International Conference on*, 1-6. IEEE.

Kieras, D. E. (2009). Model-based evaluation. *The human-computer interaction: Development process*, 294-310.

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*, 391-438.

Kuchhal, P. (2014). Ameliorating the image matching algorithm of Sikuli using Artificial Neural Networks. *International Journal of Computer Science & Communication, 5*(1), 1-4.

Laird, J. E. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Lohse, G. L. (1997). Models of graphical perception. In M. Helander, T. K. Landauer & P. Prabhu (Eds.), *Handbook of Human-Computer Interaction* (pp. 107-135). Amsterdam: Elsevier Science B. V.

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

Paik, J., Kim, J. W., Ritter, F. E., & Reitter, D. (2015). Predicting user performance and learning in human-computer interaction with the Herbal compiler. *ACM Transactions on Computer-Human Interaction, 22*(5), Article No.: 25.

Pew, R. W., & Mavor, A. S. (Eds.). (2007). *Human-system integration in the system development process: A new look*. Washington, DC: National Academy Press.

Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. M. (2000). Supporting cognitive models as users. *ACM Transactions on Computer-Human Interaction, 7*(2), 141-173.

Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. M. (2001). User interface evaluation: How cognitive models can help. In J. Carroll (Ed.), *Human-Computer Interaction in the New Millenium* (pp. 125-147). Reading, MA: Addison-Wesley.

Ritter, F. E., Kukreja, U., & St. Amant, R. (2007). Including a model of visual processing with a cognitive architecture to model a simple teleoperation task. *Journal of Cognitive Engineering and Decision Making, 1*(2), 121-147.

Ritter, F. E., & Wood, A. B. (2005). Dismal: A spreadsheet for sequential data analysis and HCI experimentation. *Behavior Research Methods, 37*(1), 71-81.

Salvucci, D. D. (2001). An integrated model of eye movements and visual encoding. *Cognitive Systems Research, 1*(4), 201-220.

Salvucci, D. D. (2006). Modeling driver behavior in a cognitive architecture. *Human Factors, 48*(3), 362-380.

Salvucci, D. D. (2009). Rapid prototyping and evaluation of in-vehicle interfaces. *ACM Transactions on Computer-Human Interaction, 16*(2), Article 9, 33 pages.

Salvucci, D. D. (2013). Integration and reuse in cognitive skill acquisition. *Cognitive Science, 37*(5), 829-860.

Soegaard, M. (2013). Interaction design foundation. *Interaction Design Foundation–Free educational materials*.

Soloway, W. L. J.-E., & Johnson, W. (1984). Intention-based diagnosis of programming errors. In *Proceedings of the 5th National Conference on Artificial Intelligence, Austin, TX*, 162-168.

St. Amant, R., Riedel, M. O., Ritter, F. E., & Reifers, A. (2005). Image processing in cognitive models with SegMan. In *Proceedings of HCI International '05*, Volume 4 - Theories Models and Processes in HCI. Paper # 1869. Erlbaum: Mahwah, NJ.

St. Amant, R., & Riedl, M. O. (2001). A perception/action substrate for cognitive modeling in HCI. *International Journal of Human-Computer Studies, 55*(1), 15-39.

Tehranchi, F., & Ritter, F. E. (2017). An eyes and hands model for cognitive architectures to interact with user interfaces. In *MAICS, The 28th Modern Artificial Intelligence and Cognitive Science Conference*, 15-20. Fort Wayne, IN: Purdue University.

Tehranchi, F., & Ritter, F. E. (2018). Using Java to Provide Cognitive Models with a Universal Way to Interact with Graphic Interfaces. In *International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation.* Washington DC, USA.

Yeh, T., Chang, T.-H., & Miller, R. C. (2009). Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22nd Annual ACM symposium on User interface software and technology*, 183-192. ACM.