Drive the Bus: Extending JSegMan to Drive a Virtual Long-Range Bus

David M. Schwartz (dms7225@psu.edu)[1]

Farnaz Tehranchi (farnaz.tehranchi@psu.edu)[2]

Frank E. Ritter (frank.ritter@psu.edu)[1]

College of Information Sciences and Technology[1]

Department of Computer Science and Engineering[2]

Penn State, University Park, PA 16802 USA

Abstract

ACT-R has been used to study human-computer interaction. By default, ACT-R models can only interact with interfaces written in Common Lisp. JSegMan has allowed ACT-R models to interact with external interfaces without modification. Currently, JSegMan has been used in conjunction with ACT-R's standard motor module, which cannot model common behaviors such as holding down keys, chording (pressing multiple keys at the same time), and multihand actions (e.g., moving the mouse with the right hand while pressing a button with the left). Extensions to ACT-R's motor module have been developed to address these issues and are included with ACT-R. Like the original motor module, the extensions can only interact with interfaces written in Common Lisp. This paper describes modifications to update JSegMan to work with ACT-R's motor extensions and demonstrates its usage by creating a model to play *Desert Bus*. Furthermore, the implication of running a model over many hours is explored.

Keywords: Cognitive architectures; ACT-R; Motor control; Chording.

Introduction

The embodied cognition-task-artifact triad states that behavior in an interactive environment is mediated by three factors: embodied cognition, the task a user is performing, and the artifact they interact with. Byrne (2001) proposes that using ACT-R (Anderson, 2007; Ritter, Tehranchi, & Oury, 2018) can assist human-computer interaction studies because ACT-R deals with the entire triad at once—the architecture handles the limits of cognition, the model encodes task knowledge, and an artifact is necessary to provide stimuli to the model and handle its output (key presses and mouse movements). However, ACT-R in its current form can only interact with special or heavily modified interfaces, making it difficult to study human-computer interaction.

JSegMan (Tehranchi & Ritter, 2018a, 2018b) offers a method of interacting with an interface external to ACT-R without modification. It detects visual features from a screenshot of the computer's display to provide ACT-R with stimuli. In addition, it allows a model's motor movements to control a computer's peripherals. However, this new level of interaction is limited to the default functionality of ACT-R's motor module and thus is limited in the behavior it can model.

By default, ACT-R is only capable of supporting serial motor action. Multiple motor commands can be queued together to simulate quick typing, but the architecture must process each keypress separately. This prevents the architecture from being able to press multiple keys at once, thereby making it impossible to type certain symbols (e.g., open and close parentheses because they require the shift key), use keyboard shortcuts, and play many video games. These issues were raised and addressed by during the development of a model to play space fortress (Bothell 2010). However, JSegMan has yet to incorporate the extended functionality. To determine how JSegMan must change, we created a model to play a simple game, *Desert Bus*.

Our experience in developing the model has led to several proposals on how to grow JSegMan. First, JSegMan should add commands (e.g., press and release) that mimic those available in the extended ACT-R motor module. Second, JSegMan can reduce its overhead (and improve model accuracy in dynamic task environments) by using ACT-R's remote procedural call interface. This work also raises questions about long-term behavior in cognitive architectures.

Background

This section discusses ACT-R's structure and various methods researchers have used to have it interact with external interfaces. Also, the game used as a task is described.

ACT-R

The ACT-R cognitive architecture (Anderson, 2007; Ritter, Tehranchi, & Oury, 2018) implements the fixed features of cognition as modules. The primary function of the architecture is controlled by the declarative and procedural modules. The declarative module manages factual memory (e.g., George Washington was the first president of the United States) encoded as chunks while the procedural module handles memory about performing actions (e.g., to turn on a computer, you have to press the start button), encoded as productions. The facts in declarative memory, actions in procedural memory, and stimuli the model sees determines how it behaves. What the model sees and how it acts within its environment are controlled by the perceptual and motor systems (spread across four modules), respectively. However, ACT-R has issues interacting with external interfaces and simulations (Schwartz & Dancy 2019; Schwartz & Ritter 2019).

ACT-R/PM

ACT-R's current perceptual and motor systems are based on ACT-R/PM (Byrne, 2001). The system assumes the model is viewing and interacting with a computer. ACT-R/PM adds four modules to the architecture: vision, motor, speech, and audition. ACT-R/PM's perceptual and motor modules have been merged into ACT-R and come as part of the standard release. This section will only discuss the vision and motor modules as the others are not pertinent to this project.

The vision module handles what an ACT-R model can see. It represents the screen as a collection of features that represent text, images, lines, buttons, etc. Features are mapped to chunks that represent where and what an object is. The visual-location buffer controls the *where* system and allows a model to query for an object's location. Once a feature is found, the model can shift its attention to it and encode the object via the *what* system controlled by the visual buffer. This creates a detailed chunk for the model to use.

The motor system provides support for using a virtual keyboard and mouse. It represents a user with two hands and allows procedural memories in a model to move the hands, mouse, and punch/peck mouse buttons and keys. The duration of hand and finger movements are estimated via Fitts' Law.

It is important to note that ACT-R/PM only works with special interfaces. ACT-R/PM was originally written in Macintosh Common Lisp (MCL) and only extracts features from interfaces created in a particular set of tools included with ACT-R/PM. ACT-R/PM has been partially generalized, allowing it to pull features from the ACT-R Graphical User Interface, across various Lisp implementations. However, the root of the problem remains—the interface still needs to be written in a compatible Lisp variant using the predefined structures.

Shortcomings of and Extensions to the Motor System. Two issues are present in ACT-R's motor system. First, ACT-R's motor system cannot perform concurrent inputs that are common in everyday computer usage. This issue is caused by state management within the motor module. The module has three states: preparation, processor, and execution. New motor commands can be queued when the preparation state is free. However, only one action can be executed at a time as the execution state handles commands serially. Furthermore, these states control actions for both hands; therefore, performing an action with one hand prevents the model from using the other. This implies that ACT-R cannot model video games that require the user to use both hands concurrently.

Second, the motor module does not support holding down keys. The motor module supports punches and pecks, each of which presses and then releases a given key. Together, these issues limit the types of interaction ACT-R can model.

These limitations prevent ACT-R models from pressing multiple keys at once—meaning the regular behavior users exhibit when typing capital letters and using keyboard shortcuts cannot be modeled. A common workaround is to assume that the model has an extended keyboard with buttons that represent chords. Thus, to give an ACT-R model the ability to use copy and paste shortcuts, dedicated buttons would be added to ACT-R's virtual keyboard to input Control-c and Control-v chords, respectively.

These weaknesses were exposed and remedied during the development of a model to play Space Fortress (Bothell 2010). Separate execution states were added per hand, allowing ACT-R to use both hands in parallel. Several motor commands were added to facilitate holding down and releasing keys such as hold-peck, hold-punch, hold-key, and release. The extended system signals both presses and releases, so new handlers were added to devices to enable them to detect key and mouse button releases. Finally, a new module, called motor-extension, was added that has two buffers that can query the activity of each hand.

Network Interfaces

Another method of getting ACT-R to interact with an external interface is via a network interface. The JSON Network Interface (JNI) (Hope, Schoelles, & Gray, 2014) allows visual objects and motor movements to be shared over a network connection. The interface generates chunks for the visual objects on screen, packs them into a JSON record, and sends it to an ACT-R model. A special module unpacks the packet and adds the information to the visicon (the list of visual features currently on screen), allowing ACT-R to work with the visual information as normal. Similarly, motor commands in ACT-R generate a packet that is sent to the interface, which can be used to update the interface's state.

New versions of ACT-R (7.6+) have incorporated similar functionality. They are based on a remote procedure call (RPC) system that allows multiple clients to request actions from a server running ACT-R. Therefore, an interface can connect to the server and send visual chunks for models to use. Additionally, the interface can watch for motor commands and act based on them.

Both JNI and ACT-R's RPC system assume an interface can be modified. The task interface must have several features added to it. First, it must manage the connection to either JNI or ACT-R's RPC server. Second, it must be able to convert visual information into visual location and encoded object chunks. Third, it must be able to simulate inputs based on those received from JNI or ACT-R. These modifications can be nontrivial and take time away from the core reason for using ACT-R, to study human cognition in a task.

Segmentation and Manipulation

Another method of providing interaction to external interfaces is by parsing the screen and manipulating inputs. Therefore, this approach aims to alleviate the issues present in ACT-R/PM and network interfaces by allowing the model to "see" what is on the screen and actually interact with it. SegMan adopted this approach (St. Amant, Riedl, Ritter, & Reifers, 2005). SegMan created visual features by taking a screenshot of the display and separating the pixels into groups based on color and location. Patterns were used to combine groups that met modeler specified criteria. Finally, patterns and groups could be parsed to identify visual features such as images, buttons, and text. In addition, SegMan could simulate mouse movement, clicks, and key presses by interacting with the operating system.

SegMan was written in C and worked with Microsoft Windows 95/98/2000/XP. In addition, it was designed to be a general programmable interface, and thus worked with several architectures including ACT-R, Soar, and EPIC. Unfortunately, the system was not maintained and over time became less usable.

JSegMan (Tehranchi & Ritter, 2018a, 2018b) is a modern implementation based on the segmentation and manipulation approach. JSegMan works separately from ACT-R, feeding visual information to it and capturing desired motor commands from it. The vision system works by taking a screenshot of the computer's display and detecting features requested by a model. Models are augmented to have memories of what an object (e.g., a button) looks like. These memories store images to search for in an interface. Finding a feature is handled by template matching—a computer vision algorithm that separates the screen into patches and compares each patch to a template (or desired image) pixel by pixel. The patch with the highest similarity to the requested memory image is returned.

Motor control is handled by interacting with the operating system. A signal representing a model's interaction (e.g., a punch or peck) is sent to JSegMan, which relays the corresponding action to the operating system.

JSegMan has shown that older models must be modified to work with real interfaces. A model designed to perform the Dismal spreadsheet task (Kim & Ritter, 2015) was modified to use JSegMan (Ritter, Tehranchi, Dancy, & Kase, in press; Tehranchi & Ritter, 2018a). The Dismal task asks subjects to compute values in a spreadsheet given a fixed set of instructions; Emacs was used to display and modify the spreadsheet. Forcing the model to really interact with the interface revealed deficiencies in the model's logic. After fixing them, the modified models performed better than the originals.

Desert Bus

The video game *Desert Bus* was used as a task during this study. *Desert Bus* was created by Dinosaur Games and published by *Gearbox Software*; it is available for free and runs on Windows machines. It was developed for a charity event. The game is based off an unreleased game of the same name designed by Penn and Teller in 1998.

Desert Bus has the player drive a bus on a straight road through the desert connecting Tucson, AZ and Las Vegas, NV. The trip takes approximately eight hours to complete one-way, at which point the player earns one point and is instructed to turn around and drive back. This process continues endlessly. All the while, the bus drifts slightly to the right. If the bus drives off the road, it is towed back to the beginning (in real-time), the trip odometer, and points are reset. The game cannot be paused. The player controls the bus with the WASD keys; W is used to accelerate, A and D turn left and right respectively, and S applies the brakes. The player can also look around with the mouse and click to open the door to the bus and turn on/off the radio. Figure 1 shows the player's view from inside the bus.

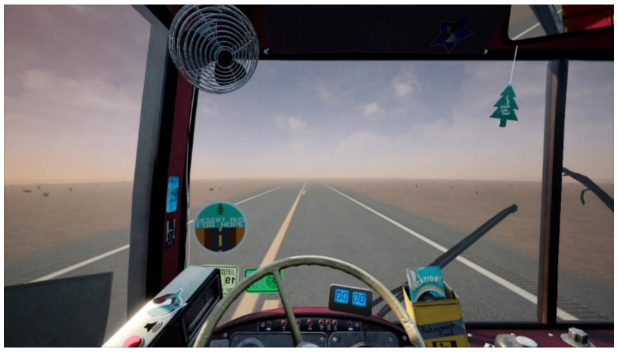

Figure 1. The player's view from inside the bus.

Model

Figure 2 shows a flowchart of the model's decision cycle. The model begins by holding down the W key to accelerate. After that, it enters a looping decision cycle where it looks for the yellow dividing line in the center of the road (Land & Horwood, 1995; Land & Lee, 1994) and uses its position to determine if the bus should be realigned. A realignment will occur if the line has drifted past 857 pixels; this is the initial position of the dividing line at the start of the game. The A key is pressed to turn the steering wheel and realign the bus. If no adjustment needs to be made, the model fires a production that symbolizes the decision to drive forward (without adjusting steering) and then restarts the decision cycle. As the game occurs in real-time, the ACT-R model also runs in real-time.

The model takes advantage of the fact that the bus will only drift to the right (causing the dividing line to move to the left). Thus, the model only has to worry about moving left or forward. A more robust model would also consider moving to the right to make up for overcompensating for the drift and ending up on the wrong side of the road. Our model does not worry about this because no other vehicles appear in the game.

JSegMan handles finding visual targets and simulating keyboard inputs for the model. Visual searches are requested at the start of the decision cycle, so the model will always know where the dividing line has drifted since the prior decision. Following the example of the Dismal model, an ACT-R device was used to detect key presses and signal JSegMan on the behavior to emulate. JSegMan does not have a persistent connection to ACT-R. Instead, a JSegMan process must be started (and run to completion) for each action. Data is passed to JSegMan via command-line arguments. Data is received from it by parsing its output stream. Furthermore, when JSegMan is running, the ACT-R model is paused.

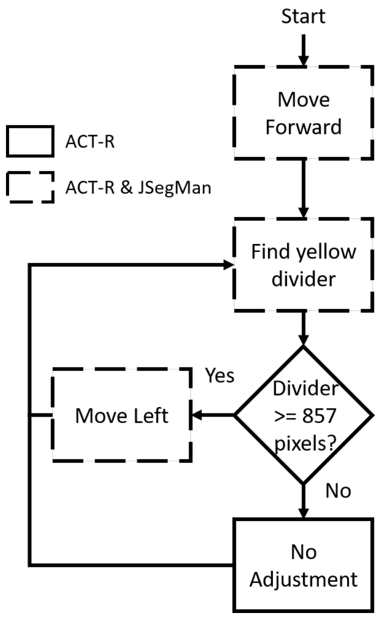

Figure 2. Flowchart of the model. Boxes with a solid border do not make use of JSegMan whereas boxes with a dashed border do. The model starts by driving forward. Then, it looks for the dividing line in the road and realigns the bus (by moving left) if the line has drifted far away.

The model only looks for the center dividing line, so it only has one template for JSegMan to look for, depicted in Figure 3. Templates in JSegMan are images, thus a screenshot of the game was used to generate the template.

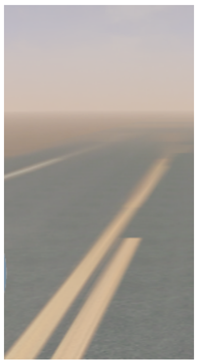

Figure 3. Visual template used for the dividing line. The template was extracted from a screenshot of the game.

Finally, the model only handles driving. The player begins the game outside of the bus and must turn around and punch a timecard before entering the vehicle. To keep the model simple, we have a player punch the timecard, enter the bus, and then we start the model. Including these steps are obvious future tasks. Nevertheless, while undertaking the drive from Arizona to Nevada, it will be one of the longest running ACT-R models.

Demonstration Observations

Unfortunately, in its current state, the model is only able to drive for about a mile before being towed back to the beginning. The model always successfully makes one adjustment. However, the adjustment made is too large; it takes the bus from the extreme right edge of the road to the extreme left of the opposite lane. After that, the model will continue driving forward until the bus drifts back into the center of the road (between the two lanes). Then the model attempts to make another adjustment and over adjusts, driving off the road to the left.

The model fails to drive for more than a mile for a multitude of reasons. First, the template for the dividing line gets mismatched. The model only uses one template to identify the location of the divider. However, this template is not always satisfactory. As the bus drifts left and right across the road, the angle of the dividing line changes. When the bus is to the right of the divider, the angle is similar to that of the template and matches are more likely to be correct. However, when the bus over adjusts and ends up on the left of the divider, the template does not match as well. Furthermore, ACT-R is unaware of the quality of a match. JSegMan is used to find objects and features on the display. However, JSegMan does not return any information about the quality of a match, but a matching request will always return a position. Thus, a feature will always be found even if it is not present, meaning ACT-R does not know when it should avoid putting the feature in the visicon.

In theory, using multiple patterns could remedy the issue. Patterns of the divider at different angles would be a proxy for where the bus is, allowing the model to determine if an adjustment is necessary. However, this process would take too long. Currently, it takes 6.01 seconds on average (n=100) to match the divider template. Furthermore, this is about the time it takes for the bus to drift from the center of the road to the rightmost edge; therefore, if the model attempted to match a second template, it would drive off the road before having the chance to make an adjustment.

Additionally, the over adjustment is an artifact created by the overhead of running external processes. JSegMan does not have support for holding keys or presses of arbitrary lengths. To make up for this, a Java program was constructed to simulate key press and release events (to mimic the signals sent by ACT-R) and is invoked just like JSegMan. This program was used to determine what the effects would be of incorporating press and release commands into JSegMan. To simulate a full key press and release this program would have to be run twice, the former sending the press signal while the latter sent the release. According to the model, an adjustment involves a rapid peck lasting for 0.08 seconds. However, on average (n=100) this mechanism takes 2.79 seconds to simulate an input.

Furthermore, the input seen by the operating system is longer than 0.08 seconds because of the time spent creating the release process. Using the newest version of ACT-R would help alleviate some issues (notably those for key presses/releases) by reducing overhead. Newer versions of ACT-R are remote procedure call based. If JSegMan is modified to be a client to ACT-R's event dispatcher, it will not need to be restarted, reducing overhead to the time it takes to send several packets (representing the command to execute). This change will require JSegMan to rely less on the device, as newer versions of ACT-R try to avoid using it. However, this should not be an issue as JSegMan will also be able to query the event dispatcher, thus it can watch for events generated by the motor module instead of the device.

Discussion and Future Work

There are some limitations to this model. It does not perform the whole task, and cannot yet drive very far. These limitations suggest changes to JSegMan and its interaction with ACT-R. Specifically, JSegMan should return information about the quality of a match and should use a persistent connection to ACT-R (especially when being used in dynamic environments) to reduce overhead. Finally, JSegMan should incorporate commands that enable models to hold down keys for arbitrary (or indefinite) lengths of time. Implementing these changes will allow JSegMan to be used in modeling more complex tasks. During our work, we also discovered several other interesting topics that can be studied with a model that can drive a *Desert Bus*.

Vigilance

The version of *Desert Bus* we used is multiplayer, allowing other players to enter the bus as passengers. Players can interact with one another by talking or throwing scraps of paper. Thus, cognitive resources are diverted away from driving. Helton and Russel (2011), showed that subjects perform worse at a target detection task when simultaneously performing a spatial or verbal working memory task. Therefore, in the future, the model can be augmented to lose vigilance while driving and interacting with passengers.

Giving Up and Physiologic Effects

Desert Bus is more a game of endurance than skill. The trip, one-way, takes about eight hours to complete and there is no end to the game; the goal is to see how far you can go. A model can play the game forever, but this is unrealistic for a person. A model can be created that weighs external influences and duties against playing and determines when to stop.

Additionally, the model can become more realistic by incorporating physiology with ACT-R/Φ (Dancy, 2013). Players can become hungry, thirsty, and/or sleep deprived while playing, causing their performance to suffer to the point that the bus runs off the road or forces the player to stop. Traditional driving models do not drive for long, so they can ignore these influences. However, ours can theoretically run forever. Adding a physiologic component to the model can reveal interactions between cognition and physiology and leads to a more robust theory of prolonged work and quitting.

Conclusion

With the advent of SegMan and JSegMan, ACT-R gained the capability to truly interact with a wide range of uninstrumented interfaces. ACT-R's motor module has evolved to enable modeling of many behaviors users may exhibit. JSegMan should evolve to make use of the extensions to ACT-R's motor module to allow models to interact with external interfaces with the same behavior as users.

Using *Desert Bus* as a task, we began exploring how to improve JSegMan and what implications our proposals had for modeling and the design of JSegMan in general. While our model did not successfully play the game for long, it yielded useful insights.

Acknowledgements

We would like to thank Jacob Oury and Matt Norris. Oury provided feedback on the paper. Norris found the remake of *Desert Bus*. We also thank everyone who attended the Virtual International Symposium on Cognitive Architectures for providing feedback on the project, particularly Dan Bothell and John Anderson. Work on this project was support by a fellowship to the first author from SMART, and a grant from ONR (N00014-15-1-2275) partially supported the second and third.

References

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

Bothell, D. (2010). Modeling Space Fortress: CMU Effort [PowerPoint slides]. *Seventeenth Annual [ACT-R] Workshop and Summer School.* Retrieved from http://act-r.psy.cmu.edu/wordpress/wp-content/themes/ACT-R/workshops/2010/talks/ICCM_SF.ppt

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. https://doi.org/10.1006/ijhc.2001.0469

Dancy, C.L (2013) ACT-RΦ; A cognitive architecture with physiology and affect. *Biologically Inspired Cognitive Architectures*, *6*(1), 40-45.

Helton, W. S., & Russell, P. N. (2011). Working memory load and the vigilance decrement. *Experimental Brain Research*, 212(3), 429–437. https://doi.org/10.1007/s00221-011-2749-1

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.

Kieras, D. E., Wood, S. D., & Meyer, D. E. (1997). Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *Transactions on Computer-Human Interaction, 4*(3), 230-275.

Kim, J. W., & Ritter, F. E. (2015). Learning, forgetting, and relearning for keystroke- and mouse-driven tasks: Relearning is important. *Human-Computer Interaction*, *30*(1), 1-33.

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

Land, M. F., & Horwood, J. (1995). Which parts of the road guide steering? *Nature, 377*(6547), 339-340.

Land, M. F., & Lee, D. N. (1994). Where we look when we steer. *Nature, 369*(6483), 742-744.

Ritter, F. E., Tehranchi, F., Dancy, C. L., & Kase, S. E. (in press). Some futures for cognitive modeling and architectures: Design patterns that you can too. *Computational and Mathematical Organization Theory*.

Ritter, F. E., Tehranchi, F., & Oury, J. D. (2018). ACT□R: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, *10*(3), e1488.

Schwartz, D. M. & Dancy, C. L. (2019). Building environments for simulation and experimentation in Malmo. *Twenty-Sixth Annual ACT-R Workshop.*

Schwartz, D. M., & Ritter, F. E. (2019). *Lessons from connecting Skirmish Sim and ACT-R/Phi* (Tech. Report No. ACS 2019-3). Applied Cognitive Science Lab, College of Information Sciences and Technology, Penn State.

St. Amant, R., Riedl, 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. Mahwah, NJ: Erlbaum.

Tehranchi, F., & Ritter, F. E. (2018a). Modeling visual search in interactive graphic interfaces: Adding visual pattern matching algorithms to ACT-R. *16th International Conference on Cognitive Modeling*, 162–167.

Tehranchi, F., & Ritter, F. E. (2018b). Using Java to provide cognitive models with a universal way to interact with graphic interfaces. *International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation*, Paper LB_15. Washington DC, USA.