A remote ACT-R device API using JSON over TCP

Ryan M. Hope*, Michael J. Schoelles and Wayne D. Gray
Cognitive Science Department
Rensselaer Polytechnic Institute

Connecting ACT-R models to software written in languages other than Common Lisp can be quite challenging. Over the years a number of attempts have been made to make this easier, some of which have been more useful and generalizable than others. Most solutions have resulted in operating system or programming language specific solutions. In the present paper we introduce the first truly cross-platform and programming language agnostic solution for connecting ACT-R to external software. We accomplish this with a simple JSON API that is transmitted over TCP.

Keywords: ACT-R, TCP, JSON, Common Lisp

ACT-R (Anderson et al., 2004) may be the most heavily used computational cognitive architecture; it has hundreds of users in different labs and universities all over the world. The architecture has excellent documentation, tutorials and there are even yearly summer schools and workshops. As a result, the entry bar for novice cognitive modelers is fairly low. While ACT-R is written in Common Lisp, users only need the most minimal knowledge of lisp syntax to write a model. You do not need to be an computer scientist to write ACT-R models. However, this assumes that a device interface already exists for the model to interact with. In ACT-R, a device interface is a layer of code that represents the external world or task environment; it is what the model interacts with. ACT-R device interfaces can be quite complex and require fairly advanced lisp programming skills to create. The fact that device interfaces need to be written in lisp not only limits ACT-Rs' potential user base but also makes it extremely difficult to get ACT-R to interact with non-lisp software.

Previous attempts to link ACT-R to external software have resulted in operating system specific solutions, programming language specific solutions, or both. Probably the first example of ACT-R being connected to external software was Gray and Sabnani (1994); they used Apple events to make ACT-R communicate with a VCR simulation written in HyperCard. While it served its purpose at the time, this solution could never generalize to operating systems other than Mac OS. The first attempt at a general solution to the problem of connecting ACT-R to external software was the cognitive model interface management system (CMIMS) by Frank E. Ritter, Baxter, Jones, and Young (2000). Communication in CMIMS was based on text strings transmitted over UNIX domain sockets. UNIX domain sockets are not supported by the ANSI Common Lisp specification so a special library called MONGSU (Ong & F. E. Ritter, 1995) was needed. In order to use MONGSU you must be able to compile C code. Needless to say, this solution is not very user friendly nor is it compatible with Windows machines.

More recently, Destefano (2010) connected ACT-R to a video game written in Python using D-BUS. D-BUS is an open-source inter-process communication (IPC) system, allowing multiple, concurrently-running computer programs to communicate with one another. Unlike the Apple events used by Gray and Sabnani, D-BUS does work on any Linux or UNIX flavor as well as Windows. However, Destefanos' D-BUS interface was written specifically for the video game he was modeling and could not be extended to other environments. Another issue with using D-BUS is that most lisp D-BUS libraries are specific to each lisp implementation making the maintenance of D-BUS based device interface difficult.

Unlike the previous solutions, Hello Java! (Büttner, 2010) is a fairly generic solution. Hello Java!, as the name suggests, is a Java package for connecting ACT-R to Java applications. Hello Java! uses text strings transmitted over TCP to communicate with ACT-R and is written as a package that other projects could use. Unfortunately, the format of the communication strings used in Hello Java! is not documented so extending the lisp side of Hello Java! to work with other programming languages like Matlab, R or Python would be difficult.

In the present paper we introduce a generic remote ACT-R device interface that work on all operating systems and is programming language agnostic. It allows any piece of soft-

*Corresponding author. Department of Cognitive Science, Rensselaer Polytechnic Institute, Carnegie Building (rm 108), 110 8th Street, Troy, NY 12180, USA. Fax: +1 518 276 8268
Email address: hoper2@rpi.edu (Ryan M. Hope)

ware, written in any language to interact with ACT-R models. The generic remote device interface, which we call the JSON Network Interface (JNI), communicates with applications over TCP using a simple JSON API. In the following sections we will describe the design of the JNI, outline the general communication flow between the JNI and the external environment, and document in detail the JSON API. Together, the following sections should provide enough information for others to successfully connect their software up to ACT-R using the JNI.

Design

The JNI is based on a client-server model where the task environment acts as the server and the JNI acts as the client. Since the environment plays the role of server, it should be running (and listening for incoming TCP connections) before the JNI attempts to connect to it. A direct benefit of this design is that the environment can be run on a computer different than the one ACT-R runs on. The environment could even be a robot assuming it has a network connection.

Unlike traditional device interfaces, the JNI is implemented as an ACT-R module[1]. This has a number of benefits. First, in ACT-R each loaded model gets it's own instances of each module. This allows the JNI to work with multiple models simultaneously since separate connection information and configuration parameters can be stored for each loaded model. Another benefit of implementing the JNI in a module is that the JNI has access to hooks and events that a traditional device does not have access to. As a result, the JNI can operate in a way that is completely transparent to the modeler. The modeler does not need to call any extra, non-ACT-R functions before or after model execution.

Communication Flow

The flow of communication between the JNI and the environment is quite simple. Once the JNI successfully connects to the environment, the JNI sends a **reset** command. The JNI will wait for a **sync** reply from this command which prevents the model from running before the environment is ready. The next command sent by the JNI to the environment, **model-run**, occurs when the model is started. Again, this command will wait for a **sync** reply from the environment before the model actually starts running. When an ACT-R model request information from the JNI, the JNI returns information stored locally in the device. It is the responsibility of the environment to update the JNI with new information when things change in the environment. Therefore, it is recommended that the environment send its first **update-display** command to the JNI before the the **sync** reply for the **model-run** command. This ensures that the models visicon is seeded before the first conflict resolution cycle. Once the model is running, any combination of the remaining commands could be sent to or from the JNI and environment.

Timing and Synchronization

The JNI supports three timing modes: asynchronous, synchronous and time-locked. In the asynchronous mode the JNI never waits for an acknowledgment from the environment after a command is sent. This mode is most useful when the environment is dynamic and the ACT-R model is running in real-time. The synchronous mode always waits for an acknowledgment from the environment after a command is sent. This mode is great for running models faster than real time in static environments. The time-locked mode is slightly different than the first two modes. In the time-locked mode, the JNI schedules a repeating event which sends the current meta-process time to the environment. This "time signal" can then be used to drive a clock in the environment. The interval of the "time signal" is an option the modeler can set. This mode is most useful for dynamic environments where ACT-R models run in real-time or faster than real-time. All timing modes support stepping though models.

Configuration

In order to configure the JNI, modelers use the ACT-R command `(sgp)`. The JNI only has three parameters:

`jni-hostname` the IP address or FQDN of the environment server (*string*)

`jni-port` the TCP port of the environment server (*integer*)

`jni-sync` the timing mode - `nil` for asynchronous, `t` for synchronous or a numeric for time-lock

The JNI will attempt to connect to an external environment when both the `jni-hostname` and `jni-port` parameters become non-**nil**.

API

All communication between the JNI and the environment is encoded in JSON (JavaScript Object Notation), a lightweight data-interchange format (Douglas Crockford, 2006). JSON is easy for humans to read and write, and it is easy for machines to parse and generate. JSON is built on two structures, *objects* and *arrays*. An object is an unordered set of name/value pairs. An object begins with { (*left brace*) and ends with } (*right brace*). Each name is followed by : (*colon*) and the name/value pairs are separated by , (*comma*). An *array* is an ordered collection of values. An array begins with [(*left bracket*) and ends with] (*right bracket*). Values are separated by , (*comma*). A *value* can be a *string* in double quotes, or a *number*, or *true* or *false* or *null*, or an *object* or an *array*. These structures can be nested.

[1]The JNI module, installation instruction and example environments and models can be obtained from the projects website: http://github.com/ryanhope/json-network-interface.

The JSON API for commands and responses sent between the JNI and environment follow a simple format. All JSON messages are based on a root JSON object that contains three keys, *model*, *method* and *params*. The value for the *model* key is the name of the current model, the value for the *method* key is the name of the command (*string*) and the value of the *params* field is JSON object. Complete JSON messages serialized to strings need to be terminated with a carraige return and linefeed.

JNI Commands

JNI commands are commands that could be sent from the JNI to the environment. Nearly all models will need to implement handlers for all JNI commands. There is one JNI command that is completely optional. The following sections describe the JNI commands an environment should expect to see.

keypress. This command is used by the JNI to inform the environment that a keyboard button has been pressed. The command takes two keyed parameters, the ASCII key code (*keycode*) and the unicode character corresponding to the button press (*unicode*). The environment should update the environment as appropriate when it receives this command. See Example 1 for an example of this command.

```
{"model": "myModel",
 "method": "keypress",
 "params": {"keycode": 65,
            "unicode": "A"}}
```
(1)

mousemotion. This command is used by the JNI to inform the environment that the mouse has moved. The command takes one keyed parameter (*loc*), the x and y location in screen coordinates of the mouse's new location as an array. When the environment receives this command it should, at the very least, store the location as mouse click events do not contain the location of the click. See Example 2 for an example of this command.

```
{"model": "myModel",
 "method": "mousemotion",
 "params": {"loc": [342, 563]}}
```
(2)

mouseclick. This command is used by the JNI to inform the environment that a mouse button has been pressed. The command takes one keyed parameter (*button*), the ID (button number) of the mouse button clicked. See Example 3 for an example of this command.

```
{"model": "myModel",
 "method": "mouseclick",
 "params": {"button": 1}}
```
(3)

speak. This command is optional and is typically not used by many models. If the environment has a text to speech (TTS) engine available, this command can be used by the JNI to convert text to an audible voice. The command takes one parameter (*message*), a string containing the word(s) to be spoken. See Example 4 for an example of this command.

```
{"model": "myModel",
 "method": "speak",
 "params": {"message": "Quick brown fox."}}
```
(4)

reset. This command is used by the JNI to inform the environment that the model has been reset. When the environment receives this command it should reset all its state as well. This command takes one keyed parameter (*time-lock*), a boolean, which when true indicates that the time-lock timing mode is being used. When this parameter is false, the synchronous or asynchronous timing mode is being used. See Example 5 for an example of this command.

```
{"model": "myModel",
 "method": "reset",
 "params": {"time-lock": false}}
```
(5)

model-run. This command is used by the JNI to inform the environment that the model is about to start running. This command takes one keyed parameter (*resume*), a boolean, which when true indicates that the model is resuming a previous run. See Example 6 for an example of this command.

```
{"model": "myModel",
 "method": "model-run",
 "params": {"resume": false}}
```
(6)

model-stop. This command is used by the JNI to inform the environment that the model has stopped running. This command takes no parameters. See Example 7 for an example of this command.

```
{"model": "myModel",
 "method": "model-stop",
 "params": {}}
```
(7)

set-mp-time. This command is used by the JNI to inform the environment of the current meta-process time. This command is only used in the time-locked timing mode and takes one keyed parameter (*time*), the current meta-process time. See Example 8 for an example of this command.

```
{"model": "myModel",
 "method": "set-mp-time",
 "params": {"time": 5.7}}
```
(8)

Environment Commands

Environment commands are commands that could be sent from the environment to the JNI during model execution. The following sections describe the available envrionment commands.

sync. This command must be sent in response to any JNI command, regardless of the current timing mode; it takes no parameters. See Example 9 for an example of this command.

```
{"model": "myModel",
 "method": "sync",
 "params": {}}
```
(9)

update-display. This command use by the environment to update the JNI's knowledge of the location of all visual objects. The command takes three keyed parameters. The first parameter (*visual-location-chunks*) takes an array of JSON encoded visual-location chunks. The second command (*visual-object-chunks*) takes an array of JSON encoded visual-object chunks equal in length to the visual-location-chunks array. The structure of a chunk in JSON is an object with two key/value pairs. The first key is *isa* and its corresponding value should be a string describing its type. The second key is *slots* and its corresponding value is another JSON object whos key/value pairs are the slot and slot values of the chunk. For slot values only, strings prefixed with a colon will be treated as a chunk. The third parameter (*clear*) is a boolean which if set to true instructs the JNI to clear the visicon before processing the new visual chunks. See Example 10 for an example of this command. This command should be called when ever visual information in the environment changes.

```
{"model": "myModel",
 "method": "update-display",
 "params": {
  "visual-location-chunks": [
   {"isa": "visual-location",
    "slots": {"screen-x": 100,
              "screen-y": 200}},
   {"isa": "visual-location",
    "slots": {"screen-x": 300,
              "screen-y": 400}}],
  "visual-object-chunks": [
    [{"isa": "visual-object",
      "slots": {"value": "hello"}},
     {"isa": "visual-object",
      "slots": {"value": ":world"}}]],
  "clear": true}}
```
(10)

trigger-reward. This command is used by the environment to reward the model. The command takes one keyed parameter (*reward*), the value of the reward. See Example 11 for an example of this command.

```
{"model": "myModel",
 "method": "trigger-reward",
 "params": {"reward": 10}}
```
(11)

set-cursor-loc. This command is used by the environment to inform the JNI of environments cursor location. The command takes one keyed parameter (*loc*), the x and y location of the cursor in screen coordinates as an array. See Example 12 for an example of this command.

```
{"model": "myModel",
 "method": "set-cursor-loc",
 "params": {"loc": [254, 355]}}
```
(12)

new-digit-sound. This command is used by the environment to create auditory chunks corresponding to digits. The command takes one keyed parameter (*digit*), a number representing the digit to be heard. See Example 13 for an example of this command.

```
{"model": "myModel",
 "method": "new-digit-sound",
 "params": {"digit": 8}}
```
(13)

new-tone-sound. This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes two keyed parameters. The first parameter (*frequency*) is a number representing the frequency of the tone to be heard. The second parameter (*duration*) is a number representing the duration of the tone to be heard. See Example 14 for an example of this command.

```
{"model": "myModel",
 "method": "new-tone-sound",
 "params": {"frequency": 440,
            "duration": 1}}
```
(14)

new-word-sound. This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes one keyed parameter (*words*), a string which is the word (or words) which will be heard. See Example 15 for an example of this command.

```
{"model": "myModel",
 "method": "new-word-sound",
 "params": {"words": "hello world"}}
```
(15)

new-other-sound. This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes four keyed parameters. The first parameter (*content*) is is the content of the sound

and can be any valid JSON value. The second parameter (*onset*) is a number which represents the amount of time between the onset and when the sound stops. The third parameter (*delay*) is a number which represents the content delay for the sound. The forth parameter (*recode*) is a number which represents the recode time for the sound. See Example 16 for an example of this command.

```
{"model": "myModel",
 "method": "new-other-sound",
 "params": {"content": "Boom!",
            "onset": 4,
            "delay": 0.1,
            "recode": 0.5}}
```
(16)

Conclusion

We feel that the JNI will greatly increase the accessibility of ACT-R as not one line of lisp needs to be written in order to connect ACT-R to other software. While the purpose of this paper was to document the language agnostic low-level API, it is possible to wrap the API in a higher level language-specific library or package. In fact, we have already made progress on this front. Available on the project website[1] is a Python package called **actr_jni** which makes connecting Twisted based Python applications to ACT-R extremely simple. In the future we plan to write (or host user submitted) high-level packages and libraries for other popular programming languages.

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological review*, *111*(4), 1036–60.

Büttner, P. (2010). "Hello Java!" Linking ACT-R 6 with a Java simulation. In D. D. Salvucci & G. Gunzelmann (Eds.), *Proceedings of the 10th international conference on cognitive modeling* (pp. 289–290). Philadelphia, PA: Drexel University.

Destefano, M. (2010). *The mechanics of multitasking: The choreography of perception, action, and cognition over 7.05 orders of magnitude* (PhD, Rensselaer Polytechnic Institute).

Douglas Crockford. (2006). *The application/json Media Type for JavaScript Object Notation (JSON)*.

Gray, W. D. & Sabnani, H. (1994). Why you can't program your VCR, or, predicting errors and performance with production system models of display-based action. In *Conference on human factors in computing systems, chi 1994* (pp. 79–80).

Ong, R. & Ritter, F. E. [F. E.]. (1995). Mechanisms for Routinely Tying Cognitive Models to Interactive Simulations. In *Hci international '95*. Osaka, Japan.

Ritter, F. E. [Frank 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.