# *Simplifying the interaction between cognitive models and task environments with the JSON Network Interface*

**Ryan M. Hope, Michael J. Schoelles & Wayne D. Gray**

**Behavior Research Methods**

e-ISSN 1554-3528

Behav Res
DOI 10.3758/s13428-013-0425-z


Springer

Behav Res
DOI 10.3758/s13428-013-0425-z

# Simplifying the interaction between cognitive models and task environments with the JSON Network Interface

**Ryan M. Hope · Michael J. Schoelles · Wayne D. Gray**



**Abstract** Process models of cognition, written in architectures such as ACT-R and EPIC, should be able to interact with the same software with which human subjects interact. By eliminating the need to simulate the experiment, this approach would simplify the modeler's effort, while ensuring that all steps required of the human are also required by the model. In practice, the difficulties of allowing one software system to interact with another present a significant barrier to any modeler who is not also skilled at this type of programming. The barrier increases if the programming language used by the modeling software differs from that used by the experimental software. The JSON Network Interface simplifies this problem for ACT-R modelers, and potentially, modelers using other systems.

**Keywords** Cognitive architecture · ACT-R · EPIC · IPC · TCP · JSON · Common Lisp · Python

Process models of cognition simulate the sequence, control, and costs of the cognitive, perceptual, and motor steps of extended sequences of behavior. The model's performance on the task is compared to the performance of humans on that same task. Depending on the modeler's focus, this comparison might include time, errors, and a detailed trace of mouse movements, keystrokes, fixations, and saccades. If the performance of the model does not match that of the humans, then the model and/or the theory underlying the model is improved/updated/rejected, and the process is repeated. Process theories of cognition allow for making tremendous progress in understanding behavior. However, using these tools requires mastering not simply the modeling formalism, but the specific programming language in which it is written. If the modeler wishes to have her model interact with experimental software, the modeler must also master the advanced art of interfacing one software system with another. If the two software systems are written in two different programming languages, this barrier becomes all but insurmountable to those without advanced training in computer science.[1]

In the sections to come, we will describe our efforts to simplify the interactions of cognitive models and task environments. The work described in this article will focus on models written with the ACT-R cognitive architecture. As a result, this work will not be immediately useful for novice ACT-R modelers; prior experience with ACT-R modeling is assumed. However, the system that we introduce is designed in such a way that it can be extended to work with many different architectures and simulation systems, which, when implemented, will facilitate cross-architecture comparisons.

## Interfacing software systems to ACT-R

The ACT-R (Anderson, Bothell, Byrne, Douglass, Lebiere & Qin, 2004) computational cognitive architecture has hundreds of users in different labs and universities across the world. As compared to other architectures, the entry-level barrier for ACT-R is quite low. It has excellent documentation and tutorials, and yearly summer schools and workshops are available. Since the perceptual–motor component was added to ACT-R 4 (Byrne & Anderson, 1998), it has been able to interact with tasks written in Common Lisp.

More specifically, the Common Lisp–based task must be a valid ACT-R device. ACT-R devices are extensions of the

R. M. Hope (✉) · M. J. Schoelles · W. D. Gray
Department of Cognitive Science, Rensselaer Polytechnic Institute, Carnegie Building (Rm 108), 110 8th Street, Troy, NY 12180, USA
e-mail: rmh3093@gmail.com

[1] See also the arguments made by Addyman and French (2012) with regard to making individual models more accessible to modelers and nonmodelers.

Published online: 13 December 2013

ACT-R framework that provide external environments with which the models can interact. For example, when an ACT-R model moves its attention to a particular visual location, ACT-R queries the active device to find out what, if any, objects are at that location. Similarly, when an ACT-R model performs motor movements such as keypresses and mouse movements,[2] those events get transmitted to the active device so that the device can update itself. An early version of this was the Ned simulation for submarine operations (Ehret, Gray & Kirschenbaum, 2000; Gray, Kirschenbaum & Ehret, 1997).

Unfortunately, few experiments and tasks are programmed in Common Lisp, and to get ACT-R models to interact with tasks written in other languages, ACT-R modelers essentially have three options. The first option is to reimplement (simulate) some portion of the original task in Common Lisp. This route was followed by Salvucci (2006), who after creating a driving simulator in C++ for human data collection, wrote another version of the same simulator in Lisp so as to interact with ACT-R.

A second option is to hard-code an implementation of the model in the language of the simulation or, more radically, to rewrite ACT-R in that language. This first path was followed by Salvucci in Distract-R; he reports (Salvucci, 2009) that only those features of ACT-R needed for the driving model were coded, and that the resulting code enabled the model and simulation to run 1,000 times faster than real time. The second path, Java ACT-R, is more recent (Salvucci, 2013) and enables the Java ACT-R model to interact directly with the Java driving simulator.

A third option is to directly connect the task software to ACT-R though an interprocess communication (IPC) mechanism. These options tend to be specific to a given operating system, programming language, or both. Gray (1995; Gray & Sabnani, 1994) pioneered this effort by connecting ACT-R through Apple events to a HyperCard simulation of a VCR. More recent examples include Schoelles's SimPilot (Schoelles & Gray, 2011, 2012) and Destefano's (2010) D-BUS connection between ACT-R and the video game Space Fortress.

D-BUS is an open-source IPC system, allowing multiple concurrently running computer programs to communicate with one another. Unlike the Apple events used by Gray and Sabnani (1994), D-BUS works on systems in any Linux or UNIX flavor, as well as on Windows. However, Destefano's (2010) D-BUS interface was written specifically for the videogame 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 to ACT-R bindings difficult.

The IPC route can be technically very challenging, but it remains the preferred route for a number of reasons. Rather than simulating the original task in Common Lisp, connecting ACT-R to the software used by the human subjects requires fewer assumptions about the equivalence of the simulation and original task. Additionally, less work is required when directly connecting ACT-R to the original task, since there are fewer opportunities for bugs, because only one software version of the task or the ACT-R architecture needs to be maintained.

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 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 & 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.

A second, and more generic, solution is Büttner's (2010) Hello Java!. Hello Java! 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.

## The JSON network interface

Our response to the problems outlined above is programming-language agnostic and works on all major operating systems. It involves three components, which we refer to collectively as the *JSON* (pronounced JAY-son) *Network Interface*, or JNI for short.[3] The first (and most important) component of the JNI is an application programming interface (API) that specifies how ACT-R and external software will communicate with each other. Second is a new ACT-R device module (referred to as the *JNI module*) that handles all of ACT-R's communications with the external environment using this new API. The third JNI component is high-level libraries (HLL) for non-Lisp programming languages that handle all communications with ACT-R. These HLLs will abstract away the complexities of the JNI API, thereby providing simple reusable libraries for researchers to use for modeling. The HLLs are not strictly required for the JNI to be useful, however.

[2] Historically, ACT-R has mostly been used to model computer-based tasks. As a result, ACT-R currently only supports hand- and finger-based movements, which are assumed to be on either a keyboard or a mouse.

[3] JSON, short for *JavaScript Object Notation*, is a lightweight data-interchange format (Crockford, 2006).

Without writing one line of Common Lisp code, the JNI API and module would allow modelers/experimenters to modify their existing experimental software to work with ACT-R. Therefore, the subsequent sections of the present article will focus primarily on describing the JNI API and the features and usage of the JNI module.

*Device module* Unlike a traditional ACT-R device, the device provided by the JNI is implemented as an ACT-R module. This property has a number of benefits. First, each ACT-R model that is loaded will get its own instance of each module (see Fig. 1). This allows for separate connection information and configuration parameters to be stored for each loaded model, thereby making it possible for multiple models to connect to the same environment and run simultaneously. Another benefit of implementing the JNI device in a module is that modules have access to ACT-R 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 JNI is based on a client–server model in which the task environment acts as the server and the JNI module 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 module 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 that ACT-R runs on. The environment could even be a robot, assuming it has a network connection.

Once the JNI module successfully connects to the environment, it sends a `reset` command. The JNI module 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 module 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 requests information from the JNI module, the JNI module returns information it has stored locally. It is the responsibility of the environment to update the JNI module with new information when the environment changes. Therefore, it is recommended that the environment send its first `display-new` command to the JNI module before the `sync` reply for the `model-run` command. This ensures that the model's visual working memory (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 module and environment. See Fig. 1 for a minimal example of the communications between the JNI module and environment.

*Timing and synchronization* The JNI supports three timing modes: asynchronous, synchronous, and time-locked. In the asynchronous mode, the JNI module never waits for an acknowledgment from the environment after a motor 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 motor command is sent. This mode is great for running models faster than real time in static environments. The time-locked mode is slightly different from the

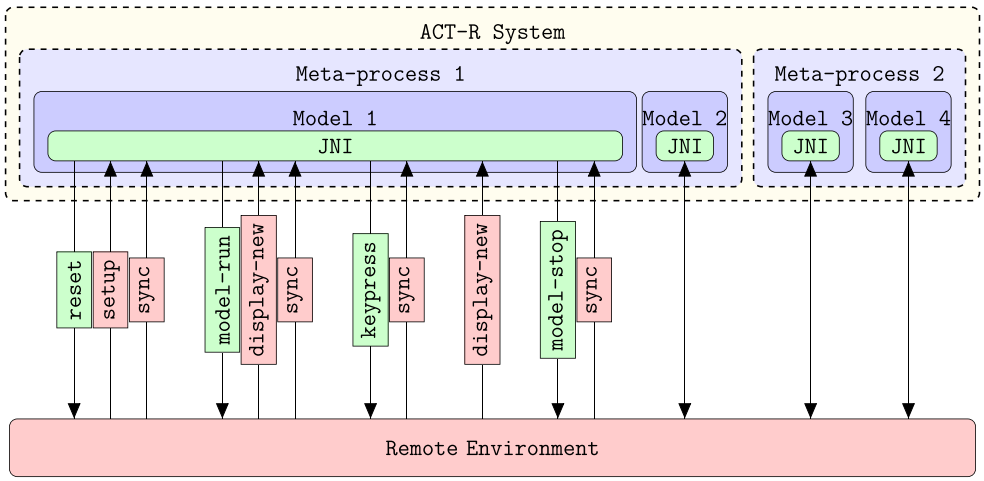


**Fig. 1** Schematic depicting how the JNI module fits into the ACT-R architecture, as well as an example of the communication flow between the JNI module and the remote environment. Nodes with rounded corners represent the system (ACT-R or remote environment) components, and square-corner nodes represent API commands. Each model has its own instance of the JNI module that communicates with the environment, independent from and in parallel with other models, via TCP. These bidirectional connections are represented by the black double-headed arrow lines for Models 2–4. For Model 1, an example sequence of API commands (ordered from left to right) is shown in place of the double-headed arrow. The colors of the API command nodes are matched to the color of the sending system component

first two modes, and is considered an advanced feature. In the time-locked mode, the JNI schedules a repeating event that 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 that the modeler can set. This mode is most useful for dynamic environments in which ACT-R models run in real time or faster than real time. Most environments will not work with this timing mode out of the box. All of the timing modes support ACT-R's stepping feature.

*Module configuration* In order to configure the JNI module, modelers use the same ACT-R command that they would use to configure any other module parameter, (`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 value for time-lock

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

## JNI API

All communication between the JNI and the environment is encoded in JSON. JSON was picked over other data-interchange formats (such as XML) because it has a much smaller grammar and maps more directly onto the data structures used in modern programming languages. JSON is also easy for humans to read and write, and 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 a: (*colon*), and the name/value pairs are separated by , (*commas*). An *array* is an ordered collection of values. An array begins with [ (*left bracket*) and ends with ] (*right bracket*). The values are separated by , (*commas*). A *value* can be a *string* in double quotes, or a *number*, or *true* or *false* or *null*, or an *object* or *array*. These structures can be nested.

The API for commands and responses sent between the JNI module and environment follow a simple pattern. 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 a JSON object. See Fig. 2 for an example of a properly formatted API command. Complete JSON messages

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

**Fig. 2** An example environment command, formatted in JSON according to the JNI API specification

serialized to strings must be terminated with a hard return and linefeed before being transmitted over TCP.

The complete set of API commands can be broken up into two groups: module commands and environment commands. *Module commands* are commands that could be sent from the JNI module to the environment. Most environments will need to implement handlers for all module commands. *Environment commands* are commands that could be sent from the environment to the JNI module during model execution. For a quick overview of the available API commands, see Table 1.

## High-level libraries

At the time of writing this article, we have completed one high-level JNI library. This library is for the Python programming language and is dependent on Twisted, an event-driven networking engine. The library, available as a Python package from the project website (see the following section, on Supplementary Materials), is called `actr6_jni`. In the future, we plan to write (or to host user-submitted) high-level packages and libraries for other popular programming languages. Although it is not complete, work has begun on creating high-level libraries for Java and MATLAB.

## Supplementary materials

Further details and documentation pertaining to the JNI API, as well as downloads for the JNI module, installation instructions, and example environments and models, can be obtained

**Table 1** Summary of the available model and environment commands supported by the JNI module

| | Command | Argument(s) | Description |
|---|---|---|---|
| Module Commands | keypress | keycode, unicode | triggered by model when a keyboard key is pressed |
| | mousemotion | loc | triggered by model when the mouse is moved |
| | mouseclick | button | triggered by model when the mouse button is clicked |
| | speak | message | triggered by model when speech is produced |
| | reset | time-lock | triggered by ACT-R when the model is reset |
| | model-run | resume | triggered by ACT-R right before a model run starts |
| | model-stop | — | triggered by ACT-R right after the model has stopped running |
| | set-mp-time | time | triggered by ACT-R as the meta-process time ticks by |
| Environment Commands | sync | — | called by environment after it handles each module command |
| | setup | width, height | called by environment when the display size changes |
| | display-new | loc-chunks, obj-chunks, clear | called by environment when all visicon chunks need replacing |
| | display-add | loc-chunk, obj-chunk | called by environment to add a chunk to the visicon |
| | display-remove | loc-chunk-name | called by environment to remove a chunk from the visicon |
| | display-update | chunks | called by environment when existing visicon chunks need updating |
| | add-dm | chunk | called by environment to add a chunk to the models declarative memory |
| | trigger-reward | reward | called by environment to reward model |
| | set-cursor-loc | loc | called by environment when the cursor location has changed |
| | new-digit-sound | digit | called by environment when a digit sound is produced |
| | new-tone-sound | frequency, duration | called by environment when a tone sound is produced |
| | new-word-sound | words | called by environment when word sounds are produced |
| | new-other-sound | content, onset, delay, recode | called by environment when a miscellaneous sound is produced |
| | trigger-event | event, args | called by environment to trigger custom lisp events |

from the project's website: http://github.com/ryanhope/json-network-interface.

## Conclusion and future work

In the present article, we have introduced a generic remote ACT-R device interface that works on all operating systems and is programming-language agnostic. It allows any piece of software, 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. Although we feel that the JNI will greatly increase the accessibility of ACT-R, since not one line of Common Lisp needs to be written in order to connect ACT-R to other software, the JNI does not need to stay specific to ACT-R. In the near future, we have plans to update the JNI API to support other cognitive architectures, starting with EPIC (Kieras & Meyer, 1997). The ultimate goal is for experimenters to be able to design one experimental task that will be compatible with multiple cognitive architectures. This would allow modelers to do something that is often talked about, yet rarely done: to compare the performance of two models running on different cognitive architectures. This will soon be feasible with the JNI.

One limitation of the JNI is that the current JNI module does not support all possible ACT-R functions that a modeler might want to call from the environment. Currently, if modelers need to call an unsupported ACT-R function, they can call `trigger-event` from their environment and implement a custom event handler for that event in their model. Once the algorithm is working, modelers can submit their custom event handlers for inclusion in future versions of the JNI. This advanced feature is described in more detail on the JNI project website, which is linked to in the previous section.

Another limitation of the JNI is related to the installation process. In order to ensure that the JNI was compatible with as many Common Lisp implementations as possible, a decision was made to use a few third-party Lisp libraries in the JNI module. These libraries need to be installed prior to using the JNI module, which we realize might make the initial use of the JNI difficult for some users. In the future, we plan on offering an installation method that includes bundled versions of all dependencies, allowing near "plug-n-play" support with ACT-R.

Finally, the current version of the JNI does not include a mechanism for transferring arbitrary data from the model to the environment. For example, an experimenter might want to include model parameters in the environment's log file. This feature will be found in the next version of the JNI.

**Author note** The work was supported by Grant No. N000141310252 to W.D.G. from the Office of Naval Research, Ray Perez, Project Officer.

## References


Addyman, C., & French, R. M. (2012). Computational modeling in cognitive science: A manifesto for change. *Topics in Cognitive Science, 4,* 332–341.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review, 111,* 1036–1060. doi:10.1037/0033-295X.111.4.1036

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: Drexel University.

Byrne, M. D., & Anderson, J. R. (1998). Perception and action. In J. R. Anderson & C. Lebiére (Eds.), *The atomic components of thought* (pp. 167–200). Hillsdale: Erlbaum.

Crockford, D. (2006). The application/json media type for JavaScript Object Notation (JSON).

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

Ehret, B. D., Gray, W. D., & Kirschenbaum, S. S. (2000). Contending with complexity: Developing and using a scaled world in applied cognitive research. *Human Factors, 42,* 8–23.

Gray, W. D. (1995). VCR-as-paradigm: A study and taxonomy of errors in an interactive task. In K. Nordby, P. Helmersen, D. J. Gilmore, & S. A. Arnesen (Eds.), *Human–computer interaction: Interact '95* (pp. 265–270). New York: Chapman & Hall.

Gray, W. D., Kirschenbaum, S. S., & Ehret, B. D. (1997). The précis of Project Nemo, phase 1: Subgoaling and subschemas for submariners. In M. G. Shafto & P. Langley (Eds.), *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society* (pp. 283–288). Hillsdale: Erlbaum.

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 *Companion to the CHI 94 Conference on Human Factors in Computing Systems* (pp. 79–80). New York: ACM Press.

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.

Ong, R., & Ritter, F. E. (1995). *Mechanisms for routinely tying cognitive models to interactive simulations*. Osaka: Paper presented at the 6th International Conference on Human–Computer Interaction (HCI International '95).

Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. M. (2000). Supporting cognitive models as users. *ACM Transactions on Computer–Human Interaction, 7,* 141–173. doi:10.1145/353485.353486

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

Salvucci, D. D. (2009). Rapid prototyping and evaluation of in-vehicle interfaces. *ACM Transactions on Computer–Human Interaction, 16,* 9. doi:10.1145/1534903.1534906

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

Schoelles, M. J., & Gray, W. D. (2011). *Cognitive modeling as a tool for improving runway safety*. Dayton: Paper presented at the 16th International Symposium on Aviation Psychology (ISAP).

Schoelles, M. J., & Gray, W. D. (2012). *Simpilot: An exploration of modeling a highly interactive task with delayed feedback in a multitasking environment*. Berlin: Paper presented at the 12th International Conference on Cognitive Modeling.