Communicating with Something Outside of Lisp: Some Available Resources and a Very Quick How-to

Dan Bothell
db30@andrew.cmu.edu

A Little Different

- We do a lot of re-implementing in Lisp
  - I don't have an example to discuss
- I have tied the internals of ACT-R 5 to an external GUI (the ACT-R Environment)
- I've helped others get it working
  - If you need help feel free to send me email

The ACT-R Environment

- GUI written in Tcl/Tk
  - available from the ACT-R web site at
    http://act-r.psy.cmu.edu/software
  - Not an "external environment" from a modeling sense
- Connects ACT-R (Lisp) through a TCP/IP socket connection to Tcl/Tk
- Works in several Lisp applications
  - ACL, MCL, LispWorks, OpenMCL, CMUCL

What can it do for you here?

- The important file is "uni-files.lisp" in the ACT-R Environment distribution
  - Lots of comments to describe what the functions do
- Contains simple functions to:
  - Open a socket
  - Write and read lines of text
  - Wait for a character
  - Spawn multiple processes/threads
- Could be used as is, or at least provide examples of the functions you need to investigate for your Lisp
- Those using MCL should consult the additional slide at the end of this presentation for some extra details

Ok, that's good, but then what?

- You have to convert the incoming state information to a representation the model can handle
- You have to convert the model's actions to something that the simulation can handle
- Now it's time for the Device Interface

The Device Interface

- An abstract representation of the world built into ACT-R/PM
  - Full details in the documentation at http://chil.rice.edu/byrne/RPM/docs/index.html
- Implemented as any Lisp class that has the appropriate methods defined

  (defclass dans-world () nil)
- Create one and let ACT-R/PM know about it

  (defvar *my-world* (make-instance 'dans-world))

  (pm-install-device *my-world*)

What are the methods necessary?

- For simplicity here, I'll assume visual info coming in and keyboard/mouse data out
  - Other interactions also predefined like auditory in and speech out
  - Also possible to define new abilities for the model
- Build-features-for
- Output-key
- Move-cursor-absolute / move-cursor-relative
- Output-click

Build-features-for

- Called to generate the visible features for the model

  (defmethod build-features-for ((world dans-world) vis-mod) …)
- Must return a list of icon-feature objects (a predefined class) or any user defined subclasses thereof
- Those features are what the model can see (the visicon)
- The feat-to-dmo method will be called to convert icon-features to visual-object chunks
  - May be user defined for subclasses
- Also need to call pm-proc-display to update the visicon when appropriate
- Collecting state info coming in and converting it to icon-features is highly dependent on the simulation

The Output Methods

- Called when the model “does something”
- Output-key
  - Called with the xy virtual keyboard position of the key
- Move-cursor-absolute / move-cursor-relative
  - Called with either an xy position or radius and angle
- Output-click
  - Called with no extra parameters
- They need to send the appropriate command to the simulation
- That’s about it then…

For those using MCL

- The file called “mcl-fix.lisp” may also be useful
- The MCL implementation of OpenTransport can be problematic out of the box
- The streams are locked on an access
  - Either you can read or write but not both
- I’ve patched the pieces to fix that
  - Had a huge improvement on the performance
  - Hasn’t caused me any troubles