Running Multiple Models and Model Task Interaction

Dan Bothell

March 24, 2016

Multiple Models

- Synchronously
  - same clock all models running together
- Asynchronously
  - separate clock per model run each individually
- Multiple instances of one model
  - parallel execution independent runs
- Multiple Lisp & ACT-R instances

Synchronous Models

- Just define each like you do a single one
  - Define-model
- Run it like before
- Each is independent of the others
  - Only share the clock

- actr7/examples/unit-1-together-1-mp.lisp

Differences in output

- Models are indicated in warnings and the trace

```
> (load "ACT-R:examples;unit-1-together-1-mp.lisp")
; Loading ACT-R:examples;unit-1-together-1-mp.lisp
    (C:\Users\db30\Desktop\actr7\examples\unit-1-together-1-mp.lisp)
#|Warning (in model SEMANTIC): Creating chunk CATEGORY with no slots |#
#|Warning (in model SEMANTIC): Creating chunk PENDING with no slots |#
#|Warning (in model SEMANTIC): Creating chunk YES with no slots |#
#|Warning (in model SEMANTIC): Creating chunk NO with no slots |#
T
> (run 1)
     0.000  COUNT     GOAL              SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
     0.000  SEMANTIC  GOAL              SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
     0.000  ADDITION  GOAL              SET-BUFFER-CHUNK GOAL SECOND-GOAL REQUESTED NIL
     0.000  COUNT     PROCEDURAL        CONFLICT-RESOLUTION
     0.000  COUNT     PROCEDURAL        PRODUCTION-SELECTED START
     0.000  COUNT     PROCEDURAL        BUFFER-READ-ACTION GOAL
     0.000  SEMANTIC  PROCEDURAL        CONFLICT-RESOLUTION
```

Working with them

- Have to indicate which model

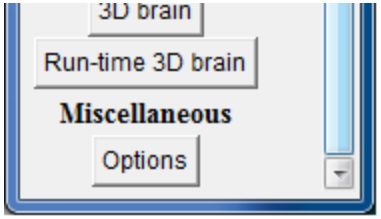
```
> (dm)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
```

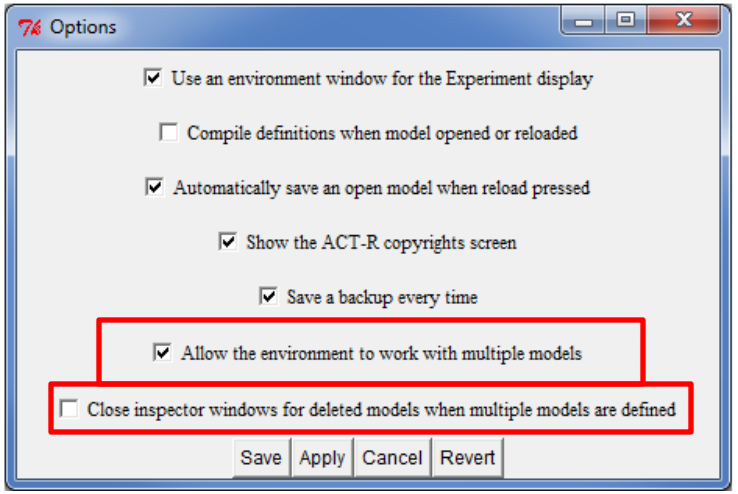
- With-model and with-model-eval commands
  - Wrap around code to indicate the model
  - With-model uses the specific name given
  - With-model-eval evaluates the expression for name

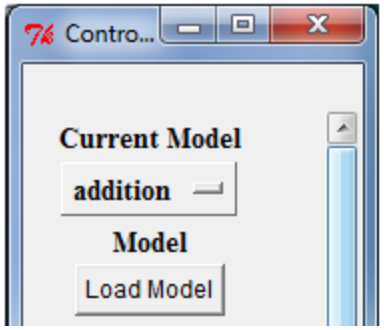
```
> (with-model count (dm))
FIRST-GOAL-0
   START  2
   END  4
   COUNT  4
…
> (let ((player 'count))
    (with-model-eval player
      (dm)))
FIRST-GOAL-0
   START  2
   END  4
   COUNT  4
…
```

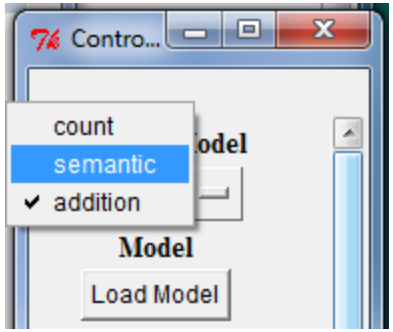
The Environment

- Enable multi-model support
- Save, stop then start
- Current model selection









- Windows indicate which model

Accessing model names

- The current model
  (current-model)

```
> (current-model)
NIL
> (with-model addition
    (current-model))
ADDITION
```

- For actions (key presses, button presses, speech)
  (model-generated-action)
  - Model name or nil
  - Better than current-model if also allowing human input

- All model names
  (mp-models)

Creating multiple identical models

- Specify the model code in a list

```
(defparameter *model-code*
   '((sgp :v t)
     (p do-nothing
        ==>
     )))
```

- Use define-model-fct to create the models

```
(define-model-fct 'model1 *model-code*)
(define-model-fct 'model2 *model-code*)
(define-model-fct 'model3 *model-code*)
```

Implementing a task

- Consider the level of abstraction
  - Necessary vs convenient
- Determine the model interactions
  - How does it perceive and act
- How will it be run
  - Recommend a continuous process
    - Zbrodoff and building stick task

Continuous running

- Use action methods and events
  - rpm-window-key-event-handler (many)
  - device-speak-string (subitizing unit 3)
  - button actions (bst unit 6)
  - Scheduled events (sperling unit 3)
- Run the model(s) until task over
  - Run-until-condition

Run-until-condition

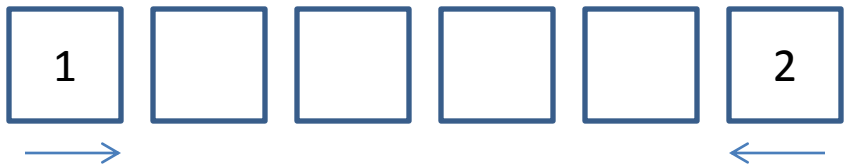
- Instead of specifying a time to run, specify a function that indicates when to stop running

```
(run-until-condition 'game-over :real-time t)
(run-until-condition (lambda () *done*))
```

- Possible gotcha
  - Model isn't doing things right and loops forever

Simple two player game

- 6 spaces in a line
- Players' pieces start at opposite ends facing each other
- Alternating turns, each can move forward 1 or 2 spaces
- A player wins when landing on or passing opponent



One game board

- One window with buttons
- Click on the button to make a move
- Current player's space is highlighted to indicate turn (red or blue)
- Players know their player color

The Window

- The window
  - Must be created in some model
  - Any model can install it as the device
  - A dummy model for “world” interactions

```
(define-model game (sgp :v nil :needs-mouse nil))

(with-model game
  (setf *window* (open-exp-window "game" ...)))

(dolist (m (mp-models))
  (with-model-eval m
     (install-device *window*)))
```

Buttons

- Every button can have an action function
  - Gets called every time the button pressed
  - Passed the button object (not usually useful)

```
(add-button-to-exp-window :window *window*
                          :x (+ 10 (* i 40)) :y 10
                          :width 35 :height 35
                          :color 'white :text ""
                          :action (lambda (b)
                                    (declare (ignore b))
                                    (pick-button index)))
```

- Can’t change features of items on screen
  - Remove and redraw buttons as they change

```
(remove-items-from-exp-window :window *window* *button1* *button2*)
```

Model

- Know its color
- Detect its turn
- Find a button to press
- Press the button
- Process the end state

Know its color

- Could create different red and blue models
- If identical models “tell” them at the start
  - Different goal chunks one approach

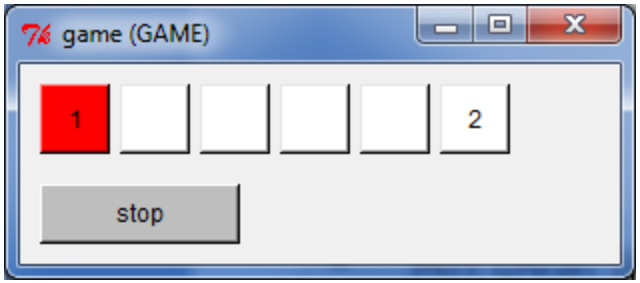
```
(with-model-eval player1
  (define-chunks (goal isa play my-color red))
  (goal-focus goal))

(with-model-eval player2
  (define-chunks (goal isa play my-color blue))
  (goal-focus goal))
```

Button features

- An oval and text co-located

Loc	Att	Kind	Value	Color	ID
(28 28)	NIL	TEXT	"1"	BLACK	VISUAL-LOCATION0
(28 28)	NIL	OVAL	OVAL	RED	VISUAL-LOCATION1
(60 75)	NIL	TEXT	"stop"	BLACK	VISUAL-LOCATION2
(60 75)	NIL	OVAL	OVAL	GRAY	VISUAL-LOCATION3
(68 28)	NIL	OVAL	OVAL	WHITE	VISUAL-LOCATION4
(108 28)	NIL	OVAL	OVAL	WHITE	VISUAL-LOCATION5
(148 28)	NIL	OVAL	OVAL	WHITE	VISUAL-LOCATION6
(188 28)	NIL	OVAL	OVAL	WHITE	VISUAL-LOCATION7
(228 28)	NIL	TEXT	"2"	BLACK	VISUAL-LOCATION8
(228 28)	NIL	OVAL	OVAL	WHITE	VISUAL-LOCATION9



The stop button

- Not there for the models
- Safety because using run-until-condition
  - A way to stop things if the models don’t work

Detecting its turn

- When its color button appears
- Use buffer stuffing of visual-location information
- Have it only stuff the critical item

```
(with-model-eval player1
  (set-visloc-default kind oval color red))

(with-model-eval player2
  (set-visloc-default kind oval - color white – color red - color gray))


(set-visloc-default-fct (list 'kind 'oval '> 'screen-x *start-x*
                              'color *my-color*))
```

Finding and pressing buttons

- Visual-location request

```
+visual-location>
   kind oval
   color white
   ...
```

- Manual move-cursor and click-mouse actions

```
+manual>
    cmd move-cursor
    loc =visual-location

 +manual>
    cmd click-mouse
```

- Possible gotcha
  - One window has one mouse cursor
  - Models need to share so set :needs-mouse to nil in ALL models

```
(sgp :needs-mouse nil)
```

Processing the end state

- Run-until-condition
  - Stops the models when result is true

```
(run-until-condition (lambda () *game-over*))
```

- Don’t want to set *game-over* immediately
  - Schedule an event to have it occur later

```
(schedule-event-relative 3 (lambda ()
                             (setf *game-over* *p2*)))
```

Pros and cons

- Single interface not too difficult to implement
- Models need to know how to play both sides
- All information available to both players

Human interaction

- Person interacting with a running model
  - Have some event in ACT-R accepting input
  - Schedule a periodic event like this might be sufficient

```
(with-model game
  (schedule-periodic-event
     .5  ;; how often in seconds
      (lambda ()
         (allow-event-manager *window*))
      :output nil))
```

    - Good enough if human interaction not critical or no ACT-R code triggered by human actions
    - Otherwise all the processing needs to be handled by the event

- Make sure to run in real-time anytime there are real windows involved

Two game boards

- Two interface windows
  - One per model
  - Both players have same perspective
    - Think they're red
    - Play left to right

Creating the windows and buttons

- Do everything within the appropriate model
  - Don’t need to specify the :window for buttons

```
(with-model-eval *p1*
    (install-device
      (open-exp-window "game" :height 100 :width 300 :visible t))

    (dotimes (i 6)
      (setf (nth i *spaces-1*)
        (let ((index i))
          (add-button-to-exp-window
                :x (+ 10 (* i 40))
                :y 10
                :width 35 :height 35
                :color (if (zerop i) 'red 'white)
                :text (if (zerop i) "1" (if (= i 5) "2" ""))
                :action (lambda (b) (pick-button b index))))))

    (proc-display))
```

Pros and Cons

- Interface code a little more complicated
- Models see same interface as either player
  - No difference between models
- Allows for hidden information
- Still have to deal with finding and using buttons

One window no buttons

- One window for both players
- Only display position number
  - Color coded by player
- Press 1 or 2 to make your move
- The game speaks the starting player's name
- A player says "done" after making a move

- Model sensitive to aural-location stuffing
- Needs to know own color and name
  - Set in goal like before

Handle the key presses

- The rpm-window-key-event-handler
  - Like most of the tutorial tasks

```
(defmethod rpm-window-key-event-handler ((device rpm-window) key)
 (let ((model (model-generated-action)))
   …))
```

Having the models talk to each other

- Use the device-speak-string method to have a model's vocal output sent to other models
  - Fake it for a human player in the key handler

```
(defmethod device-speak-string ((device rpm-window) string)
  (dolist (model (mp-models))
    (unless (eq model (current-model)) ; for everyone but current

       (with-model-eval model

         ;; Generate a word sound with the location
         ;; indicating the current model as the speaker.

        (new-word-sound string (mp-time) (current-model))))))
```

- Another example in actr7/examples/multi-model-talking.lisp

Configuring aural-location stuffing

- Set-audloc-default
  - Very similar to set-visloc-default

Model1

Sound event	Att	Detectable	Kind	Content	location	onset	offset	Sound ID
AUDIO-EVENT0	T	T	WORD	"MODEL1"	START	0	300	WORD0
AUDIO-EVENT1	NIL	NIL	WORD	"done"	SELF	1400	1600	WORD1

Model2

Sound event	Att	Detectable	Kind	Content	location	onset	offset	Sound ID
AUDIO-EVENT0	T	T	WORD	"MODEL1"	START	0	300	WORD0
AUDIO-EVENT1	NIL	NIL	WORD	"done"	MODEL1	1400	1600	WORD1

(set-audloc-default - location self :attended nil)

Processing aural info

```
(p hear-something
    =aural-location>
    ?aural>
     state free
    ==>
    +aural>
     event =aural-location)
```

```
(p my-turn-to-start
    =goal>
      my-name =name
    =aural>
      content =name
  ==>
    =goal>
      state move)

(p other-player-done
    =goal>
    =aural>
      content "done"
  ==>
    =goal>
      state move)
```

Talking

```
(p say-done
      =goal>
        state say-done
      ?vocal>
        state free
      ?manual>
        state free
    ==>
      +vocal>
        cmd speak
        string "done"
      =goal>
        state nil)
```

Pros and cons

- Interface code easier with 1 window, no buttons, and simple method for models to talk to each other
- Model is a little more complicated because it needs to handle both visual and aural percepts
  - Doesn’t need to deal with buttons
- Single interface doesn’t allow for hidden info

Two windows no buttons

- Each model has its own window
- All it sees is the opponent's position
  - Keeps track of its own location
- Press keys to act
- Hears the starting player's name at start

- Needs to know starting position and name
  - Again set in the initial goal chunk

Visual Scene Change

- Don’t use visual-location stuffing
  - Not always convenient if location info important
- Query for a scene change

```
?visual> scene-change t
```

- True if proportional change more than threshold

```
(sgp :scene-change-threshold #)
```

  - Default is .25
- Send a clear request to visual to reset the notice
  - Only true briefly, :visual-onset-span which defaults to .5 seconds

Detecting other player moved

```
(p do-something
    ?visual>
      scene-change t
      state free
    =goal>
      state play
  ==>
    +visual>
      cmd clear

    +visual-location>
    =goal>
      state attend)
```

Pros and cons

- Simple interface implementation
- Hidden information possible
- Model needs visual and aural attention
- Model doesn’t have to be ready for visual-location buffer stuffing

No windows

- Like the 1-hit blackjack task
- All information provided in a goal chunk
- Each player gets a new goal when it needs to make a move and when game over
- Use a !eval! to indicate its action
- Game information represented player specific
  - my-position and opp-position slots

Creating new goals

- Create a chunk without a specific name and use the returned name to set goal

```
(with-model-eval *p2*
  (goal-focus-fct
    (car (define-chunks-fct
           (list (list 'isa 'play 'my-position *p2-position*
                       'opp-position *p1-position*
                       'state 'move))))))
```

!eval! action function

- Don’t do things right away
- Schedule them to happen after everything else the model needs to do now

```
(p 2-step
  ...
==>
  !eval! (make-move 2)
  =goal>
    state wait
  ...)

(defun make-move (dist)
  (with-model-eval *current-player*
    (schedule-event-now 'process-move
                        :params (list dist)
                        :priority :min)))
```

Safety Check in run

- No stop button to use
- Add a model timeout or other safety check

```
(run-until-condition
     (lambda () (or *game-over* (> (mp-time) 100))))
```

Pros and cons

- No interface code
- Interaction code not too hard to implement
- Model fairly simple since states set externally
- Lots of goal chunks could be an issue for learning
- No real timing information

No window updating information

- State info in goal chunk
  - Player position and game state
- Board positions in imaginal chunk
  - Same for each player
- Use a !eval! action to make move

Buffer Chunks

Model1

```
Goal
    STATE  MOVE
    ME  P1
    OPP  P2

Imaginal
    P1  0
    P2  5
```

Model2

```
Goal
    ME  P2
    OPP  P1

Imaginal
    P1  0
    P2  5
```

Accessing the imaginal info

- Use a variable to access the correct slot

```
(p my-turn
   =goal>
     me =me
     state move
   =imaginal>
     =me =val
 ==>
 !output! (I am at position =val)
 …)
```

Schedule the buffer changes

- Setting initial chunks

```
(with-model-eval player1
  (define-chunks
     (goal isa play me p1 opp p2 state move)
     (game isa board p1 0 p2 5))
  (goal-focus goal)
  (schedule-set-buffer-chunk 'imaginal 'game 0))
```

- Modifying information

```
(with-model-eval *p2*
 (schedule-mod-buffer-chunk 'goal (list 'state 'move) 0)
 (schedule-mod-buffer-chunk 'imaginal
   (list 'p1 *p1-position* 'p2 *p2-position*) 0))))
```

Pros and cons

- No interface code
- Interaction code fairly easy to implement
- Model fairly easy to implement
  - Needs to be careful with buffer chunks
- No real timing information

Asynchronous models example

- Have to run the models individually
- Won’t work with the Environment
- Don’t recommend it

Saving a model

- An extra for saving chunks and productions
  actr7/extras/save-model/save-chunks-and-productions.lisp

- Call the save-chunks-and-productions function
  - Requires the name of a file to save

```
(save-chunks-and-productions "class/model/save.lisp")
```

- Writes out a model file with declarative and procedural information
  - Can be loaded
  - May not be sufficient

- Current version doesn't work with multiple models
  - New version with the example games that works inside a with-model