Multiple Models

- More than one model can be defined
  - Define-model
  - Clear-all removes all models
- All models run when ACT-R runs
  - same clock
- Each is independent of the others
  - Only share the clock

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 NIL
     0.000  SEMANTIC  GOAL                  SET-BUFFER-CHUNK GOAL G1 NIL
     0.000  ADDITION  GOAL                  SET-BUFFER-CHUNK GOAL SECOND-GOAL 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 |#
```

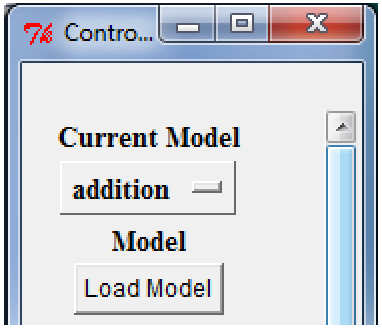
- Lisp: current-model, with-model, and with-model-eval commands
  - With-model uses the specific name given
  - With-model-eval evaluates the expression for name
- Python: actr.current_model and actr.set_current_model

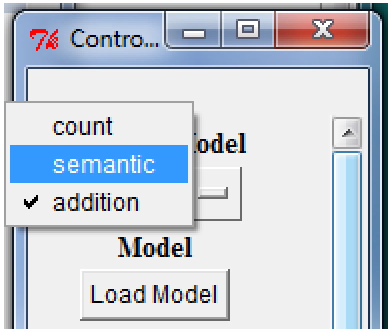
```
? (with-model count (dm))
FIRST-GOAL-0
   START  2
   END  4
   COUNT  4
…
? (let ((model 'count))
    (with-model-eval model
      (dm)))
…
? (with-model count
    (current-model))
COUNT
```

```
>>> actr.set_current_model('count')
>>> actr.dm()
FIRST-GOAL-0
   START  2
   END  4
   COUNT  4
…
>>> actr.current_model()
'count'
```

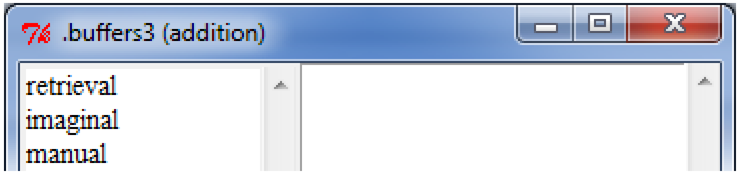
The Environment

- Current model selection





- Windows indicate which model



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 and convenient
- Determine the model interactions
  - How does it perceive and act
- How will it be run

Continuous running

- Easy to stop and start
- Monitor for the actions
  - Output-key (zbrodoff unit 4)
  - Output-speech (subitizing unit 3)
  - Move-cursor & Click-mouse
- Scheduled events
  - Sperling unit 3
- Use button action functions
  - Bst unit 6
- Run the model(s) until task over

Run-until-condition

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

```
? (run-until-condition 'game-over)

> actr.run_until_condition("done")
```

- Possible gotcha
  - Model isn’t doing things right and loops forever
  - Add a time limit and/or a safety stop button

```
(defun game-over (time)
  (or *safety-stop* *game-over* (> time 1000000)))

def is_game_over(time):
    return (safety_stop or game_over or (time > 1000000))
```

- Downside
  - Run-until-condition can be slower because it calls the test fn a lot

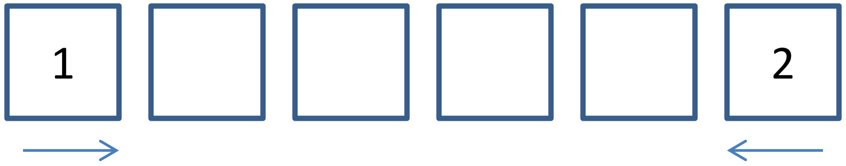
Schedule a break event to stop ACT-R

- Run model(s) very long time
- When task complete schedule a break event
  - Probably also still want a safety stop

```
(schedule-break-relative 0)
actr.schedule_break_relative(0)
```

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



Many ways to implement

- No interface, goal modification
- No window, but visual information
- One window both players press buttons
- One window players speak the move
- Two windows, one per model

No windows explicit goal chunks

- Like the 1-hit blackjack task
- All information provided in a goal chunk
  - Including which player
- Each player gets a new goal when it needs to make a move and when game over
- Use a !eval! to indicate its action

Pros and cons

- No GUI code
- Model states set externally
- Chunk manipulation code
  - Modifying while model using it could be confusing

No window but still visual info

- Provide custom visicon features directly
  - Can contain any info needed
- Model will press keys to indicate move

Add/delete/modify visicon-features

- Specify the visual-location and visual object chunks’ slot values
- Location must have position info
  - Screen-x and screen-y (default)
- Anything else is up to the modeler

Adding visicon feature example

- Directly specify the chunks for the location and object
  - Single value goes to both chunks (except position info)
  - Two values: first goes to location chunk and second to object chunk
    - Visicon holds object’s value – searchable in a visual-location request

```
(add-visicon-features `(isa (player-loc player) screen-x 0 screen-y 0 name ,player1 position (nil 0) turn (nil t))
                      `(isa (player-loc player) screen-x 100 screen-y 0 name ,player2 position (nil 5)))
```

```
actr.add_visicon_features(['isa',['player-loc','player'],'screen-x',0,'screen-y',0,'name',player1,'position',[False,p1_position], 'turn',[False,True]],
                          ['isa',['player-loc','player'], 'screen-x',100,'screen-y',0,'name',player2, 'position',[False,p2_position]])
```

```
(chunk-type (player-loc (:include visual-location)) name position turn result)
(chunk-type (player (:include visual-object)) name position turn result)
```

```
Name         Att  Loc           Kind    Position  Name    Size  Turn
-----------  ---  ------------  ------  --------  ------  ----  ----
PLAYER-LOC0  NEW  (  0 0 1080)  PLAYER  0         MODEL1  1.0   T
PLAYER-LOC1  NEW  (100 0 1080)  PLAYER  5         MODEL2  1.0
```

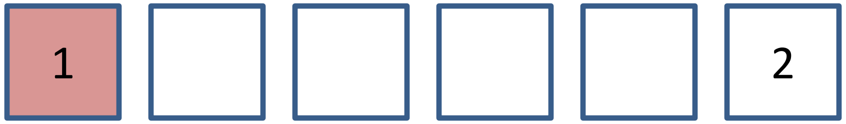
Pros and cons

- No ACT-R GUI code
- May need interface device
  - Can add to model definition

  `(install-device '("motor" "keyboard"))`
- Need to manage visicon features
  - Vision module handles “safe” updating of visicon

One game board

- One window with buttons
  - – Both models install the same device
- Click on the button to make a move
- Current player’s space is highlighted to indicate turn (red or blue)



Model

- Needs to 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))
```

Detecting its turn

- When its color button appears
- Use buffer stuffing of visual-location information
- Have it only stuff the critical item
  - Set-visloc-defaults (unit 3 code document)
    - Only available in model definition

```
(set-visloc-default kind oval - color white - color blue)
```

Finding and pressing buttons

- Unit 6 Bst task

- Visual-location request

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

- Manual move-cursor and click-mouse actions

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

+manual>
   cmd click-mouse
```

Processing the end state

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

- Schedule an event to give the model(s) time to process it

Pros and cons

- Single GUI not too difficult to implement
- Specific models or need to know how to play both sides
- All information available to both players

Two game boards

- Two interface windows
  - One per model
- Provide egocentric perspective

Pros and Cons

- Interface code a little more complicated
- Models see same interface as either player
- Allows for hidden information

One window no buttons

- One window for both players
- Only display position number
  - Color coded by player
- Say “one” or “two” to make the move
- The game speaks the starting player’s name

Having the models talk to each other

- Speaking unit 3 subitize task

```
+vocal>
    cmd       speak
    string    ...
```

- Output-speech monitor can create sounds for other models
  - New-word-sound / new_word_sound
    - Similar to new-tone-sound in unit 3 sperling code
- Set-audloc-default
  - Similar to set-visloc-default
  - Own speech has location of self

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

Pros and cons

- Simple GUI code
- Model state driven by percepts
- Processing speech in model and task may be difficult