Extending ACT-R 6.0

Dan Bothell

July 17, 2007

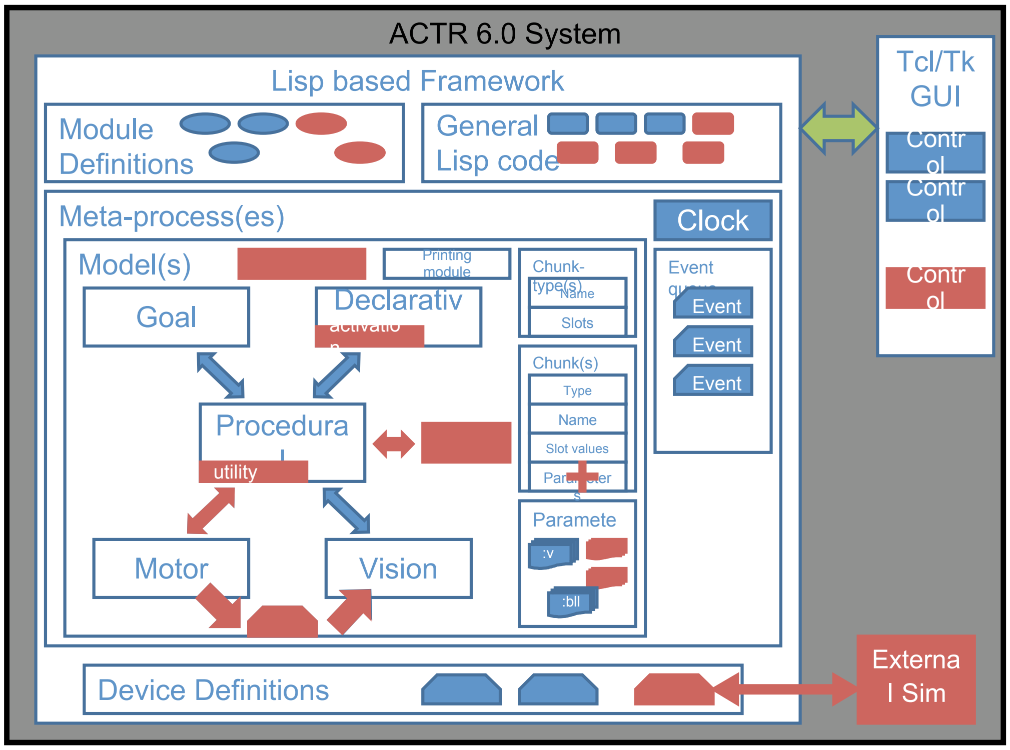
Outline

- Software overview
- Extensions
- Creating Devices
- Defining modules

Extending the system

- Changes that you can make without touching the existing code base
- Several directories in the source tree will have all the *.lisp files compiled and loaded automatically
- The Environment GUI also loads any *.tcl file in its dialogs directory
- For more details on most of this see
  - Reference manual
  - Framework API doc

ACT-R software extensions



Devices

- By default a device expected to accept/provide
  - Keyboard output (from model)
  - One button mouse control
    - Movement (from model)
    - Click output (from model)
    - Report its current position (to model)
  - Speech output (from model)
  - Visual-location information (to model)
    - features
  - Visual object information (to model)
    - Given one of the features

What *is* a device?

- Anything can be a device
  - Basic types
    - numbers, strings, lists, symbols, etc.
  - Structures
  - CLOS classes
- Whatever value is passed to install-device will be considered the current device for the model
- What makes it a **valid** device is that it have the appropriate methods defined for it
- It is the methods that allow the device to work

What is a method?

- Simplified view
- A function which specifies the “type” of some of its arguments
- Can use the same method name with different types
  - Lisp calls the right one
- Adding new ones doesn’t disturb existing ones

```
(defmethod what-is-it ((x number))
   (pprint "It is a number"))

(defmethod what-is-it ((x string))
   (pprint "It is a string"))

(defmethod what-is-it ((x list))
   (pprint "It is a list"))


> (what-is-it 2)
"It is a number"

> (what-is-it nil)
"It is a list"

> (what-is-it "foo")
"It is a string"
```

Device’s motor interface methods

- Device-handle-keypress
- Device-handle-click
- Device-move-cursor-to
- Get-mouse-coordinates

Device-handle-keypress

- Called whenever the motor module makes a keypress action
- Passed two parameters
  - – The current device
  - – A character of the key pressed by the model
- The return value is ignored

Device-handle-click

- Called whenever the model clicks the mouse
- Passed one parameter
  - – The current device
- The return value is ignored

Device-move-cursor-to

- Called whenever the model moves the mouse
- Passed two parameters
  – The current device
  – A two element vector containing the x,y position
- The return value is ignored

Get-mouse-coordinates

- Called whenever the motor module needs to know the mouse’s location
- Passed one value
  – The current device
- Must return a two element vector of x,y position

Example for a simple device

- Assume we want to use a list as a device
  - More as to why later
- Don't have any real actions
  - Just print out the information
  - Best to provide the methods for the device even if not used by the model to be safe
- Note: using globals not advised in the context of multiple models or devices
  - Keep that info "in" the device

```
(defvar *mouse-pos* (vector 0 0))

(defmethod device-move-cursor-to ((device list) loc)
  (model-output "Model moved mouse to ~A" loc)
  (setf *mouse-pos* loc))

(defmethod get-mouse-coordinates ((device list))
  *mouse-pos*)

(defmethod device-handle-click ((device list))
  (model-output "Model clicked the mouse"))

(defmethod device-handle-keypress ((device list) key)
  (model-output "Model pressed key ~c" key))
```

Device's speech interface method

- Device-speak-string
  - Called whenever the model does a speak action
  - Two parameters
    - Current device
    - A string of the model's speech output
  - Return value ignored

```
(defmethod device-speak-string ((device list) string)
  (model-output "Model said ~s" string))
```

Device’s vision interface methods

- B[illegible]eatu[illegible]s-for
- F[illegible]at-t[illegible]dmo
- C[illegible]or-to[illegible]ture

- Build-vis-locs-for
- Cursor-to-vis-loc
- Vis-loc-to-obj

Build-vis-locs-for

- Called as part of proc-display
- Passed two values
  - The current device
  - The current vision module
- Must return a list of chunks which are “isa visual-location”
  - any subtype of visual-location

- Those chunks are copied and constitute the model’s visicon

Cursor-to-vis-loc

- Called as part of proc-display and when the mouse is moved
- Passed one value
  - The current device
- Must return a chunk which is a subtype of visual-location or nil

- Called to generate the feature for the mouse cursor for the visicon
  - can update outside of a proc-display

Vis-loc-to-obj

- Called when the model moves attention to a visual feature
- Passed two values
  - The current device
  - The visual-location chunk of the feature
    - One of the ones that build-vis-locs-for provided
- Must return a chunk which “isa visual-object” or return nil

- That chunk will be the one copied into the visual buffer when encoding completes

The list device

- Before looking at examples of the methods we need to define exactly how our list is to be used as the device
- Our device list will consist of pairs where the car of the pair is a visual-location chunk and the cdr is the visual-object chunk which corresponds to it
- For simplicity, we'll assume that the model will not need a feature for the mouse cursor

The visual methods for the list device

```
(defmethod cursor-to-vis-loc ((device list))
  nil)


(defmethod build-vis-locs-for ((device list) vismod)
  (mapcar 'car device))


(defmethod vis-loc-to-obj ((device list) vis-loc)
  (cdr (assoc vis-loc device)))
```

Actually creating and using one

- Create some new chunk-types to use
- Create the chunks
- Build the list
- Install the device
- Call proc-display
- Run the model

```
(defun do-experiment ()
  (chunk-type (polygon-feature (:include visual-location)) regular)
  (chunk-type (polygon (:include visual-object)) sides)
  (chunk-type (square (:include polygon)) (sides 4))

  (let* ((visual-location-chunks
           (define-chunks
             (isa visual-location screen-x 10 screen-y 20 kind oval
                             value oval height 10 width 40 color blue)
             (isa polygon-feature screen-x 10 screen-y 50 kind square
                             value square height 30 width 30 color red regular
     t)
             (isa polygon-feature screen-x 50 screen-y 50 kind polygon
                             value polygon height 50 width 40 color blue)
             (isa polygon-feature screen-x 90 screen-y 70 kind polygon
                             value polygon height 50 width 45 color green)))
         (visual-object-chunks
           (define-chunks
             (isa oval value "The oval" height 10 width 40 color blue)
             (isa square value "square" height 30 width 30 color red)
             (isa polygon value "Poly1" height 20 width 40 color blue sides 7)
             (isa polygon value "Poly2" height 50 width 45 color green sides
     5)))
         (the-device (pairlis visual-location-chunks visual-object-chunks)))

    (install-device the-device)
    (proc-display)
    (run 10)))
```

Define-chunks

- The general command to create chunks
- Just creates the chunks as specified and returns the list of their names
- A lot like add-dm (which you should know)
  - The chunk descriptions are the same for both
  - Add-dm actually uses define-chunks
  - Add-dm explicitly places those chunks into the declarative memory of the model

More on creating devices

- That was a simple example
  - Pre-generated features and objects
  - Still probably useful for some modeling work
- More examples available with the new-vision distribution
  - Little more advanced
  - Cover some additional components available

What is a module?

- It can be any “thing” you want
  - A lot like a device
  - Instead of methods it’s defined by a set of hook functions
    - A little more freedom
    - Possibly a little less intimidating
- It is a component of the system
- Available to all models
- No real restrictions on what it can do
  - New cognitive component
  - New tracing/debugging tool
  - Modifier of parameters for other modules
  - An interface to some external system

Basic requirements of a module

- A name
  - Any symbol not already naming a module
- Version and documentation strings
- Any parameters the module requires
- Functions which
  - Create a new instance when a model is created
  - Reset an instance when the model is reset
  - Set/return the parameters' values for the module
  - Delete the instance when the model is deleted
- Interface to procedural module
  - Buffer(s)
  - Request function
  - Query function

How to create one

- Specify the definition of the module using the define-module command

```
(defun define-module-fct (name buffers params-list
                          &key version documentation
                          creation reset params delete
                          request query …)
```

Some things to note

- Must be defined outside of any models
- Recommended that it happen in a file which gets loaded with the main system
- Cannot redefine one after it’s created
  - Once it’s created you need to undefine it if you want to change the definition
  - Undefine-module

Our new module: Demo

- Similar to the imaginal module
- Will have two buffers
  - Create
    - Requests create new chunks for the buffer
    - Has a time cost (when :esc set to t)
  - Output
    - Takes requests which just print out a value in the trace
    - Happens immediately
- Has one parameter
  - :create-delay which specifies how long to take in creating the chunk when :esc is set to t

What will our module instance be

- Pick something to use as an instance for the module
  - Each model will have its own instance of the module
- Important when there are multiple models
- For this one we'll use a structure
  - two slots to hold the parameter values
  - One slot to keep track of whether the module is busy

```
(defstruct demo-module delay esc busy)
```

Start defining the module

- Specify
  - Name
  - Version
  - Documentation
- Version and doc
  - shown by mp-print-versions command
  - Printed after initial system loading

```
(define-module-fct 'demo ???? ????
   :version "1.0a1"
   :documentation
     "Demo module for ICCM tutorial"
…)
```

```
>(mp-print-versions)
ACT-R Version Information:
Framework      : 1.2
…
DEMO           : 1.0a1 Demo module …
…
```

Next thing is the buffers

- They're maintained by the system
  - Module doesn't need to do anything other than specify their names
    - must be unique among the buffers

```
(define-module-fct 'demo '(create output) ????
   :version "1.0a1"
   :documentation "Demo module for ICCM tutorial"
…)
```

- Things other than the name can be specified
  - See the manuals (API framework doc right

Parameters

- Defined in conjunction with the module
- The module is responsible for maintaining the value
- The system takes care of
  - Initializing
  - making them available to the user
  - Checking values for validity
- Users access them through the sgp command
  - Sgp calls the module's param function to get the current value when requested
  - Sgp calls the module's param function to set a new value when a valid value is provided by the user

Defining Parameters

- A parameter consists of
  - A name
    - Must be a keyword
  - Documentation
    - A string
  - Default-value
    - The initial value to set upon model reset
  - Function to check for validity
    - Used by sgp to test user values
    - passed the user's requested value
    - If it returns non-nil the value is considered valid
  - Warning to print when invalid value given
    - String that goes at the end of this warning message:

```
#|Warning: Parameter name cannot take value 3 because it must be warning. |#
```

  - Indication of ownership
    - T or nil
    - The owner is called to get the current value
    - Non-owners are notified when the parameter changes

```
(defun define-parameter (param-name &key (documentation "") (default-value nil)
                                         (valid-test nil) (warning "") (owner t))
```

How that ties into the module

- The third parameter to define-module is a list of parameters for the module
- Define-parameter just creates an abstract parameter
- It needs to be part of a module definition to become available to the model

```
(define-module-fct 'demo '(create output)
   (list (define-parameter :create-delay
               :documentation
                 "time to create the chunk for the demo module"
               :default-value .1
               :valid-test (lambda (x)
                              (and (numberp x) (>= x 0)))
               :warning "Non-negative number"
               :owner t)
         (define-parameter :esc :owner nil))
   :version "1.0a1"
   :documentation "Demo module for ICCM tutorial"
…)
```

The params function

- It will be called by sgp with two values
  - The current model's instance of the module
  - The second will be either
    - The parameter name if it is asking for the current value
    - A cons of the parameter name and the new value to set
- Should return the current value if it is the owning module
- Also called during model reset to set the default value
  - After the module's reset function*

Basic params function operation

- Save our parameter when the user changes it
- Note the value of :esc when it's changed
- Return the :create-delay value when the user requests it

```
(defun demo-module-params (demo param)
  (if (consp param)
    (case (car param)
      (:create-delay
       (setf (demo-module-delay demo)
             (cdr param)))
      (:esc
       (setf (demo-module-esc demo)
             (cdr param))))

    (case param
      (:create-delay
       (demo-module-delay demo)))))
```

Progress now

```
(define-module-fct 'demo '(create output)
   (list (define-parameter :create-delay
               :documentation
                 "time to create the chunk for the demo module"
               :default-value .1
               :valid-test (lambda (x)
                               (and (numberp x) (>= x 0)))
               :warning "Non-negative number"
               :owner t)
         (define-parameter :esc :owner nil))
   :version "1.0a1"
   :documentation "Demo module for ICCM tutorial"
   :params 'demo-module-params
…)
```

- We now need the other basic module interface functions

Creation function

- The module’s creation function will be called every time a new model is defined
- It will be passed one parameter which is the name of the new model
- It must return a new instance of the module to use for that model

```
(defun create-demo-module (model-name)
  (make-demo-module))
```

Delete function

- The module’s delete function will be called when the model is deleted
  - At a clear-all
  - When delete-model called directly
- It will be passed one parameter which is the instance of the module in the current model
- The return value is ignored
- Perform any necessary cleanup
- Not mandatory that one be provided

```
(defun delete-demo-module (demo)
  )
```

Reset function

- The module’s reset function will be called
  - after the initial creation
  - every time a model is reset
- It will be passed one parameter which is the instance of the module in the current model
- The return value is ignored
- Perform any necessary initialization
- Not mandatory that one be provided
- We need to create the chunk-type for our output request
  - That way it will always be available (not depending on the modeler to add it)

```
(defun reset-demo-module (demo)
  (chunk-type demo-output value))
```

Progress

- Add those functions to the definition
- All that's left are the functions for the buffers

```
(define-module-fct 'demo
                   '(create output)
  (list (define-parameter :create-delay
               :documentation
                  "time to create the chunk for the demo
   module"
               :default-value .1
               :valid-test (lambda (x)
                              (and (numberp x) (>= x 0)))
               :warning "Non-negative number"
               :owner t)
        (define-parameter :esc :owner nil))

   :version "1.0a1"
   :documentation "Demo module for ICCM tutorial"
   :creation 'create-demo-module
   :reset 'reset-demo-module
   :delete 'delete-demo-module
   :params 'demo-module-params
)
```

Tying the module into the productions

- To allow the productions (or other modules) to interact with your module you need to add buffer(s) to your module
- Then specify the functions which will handle the requests and the queries

The queries

- Represent instant “checks” of the module
  - Specified with a slot and a value

```
?visual> state free
```

- Should return immediately with true or false
  - Nil means false anything else is true
- Must accept queries for the module’s state
  - State busy
  - State free
  - State error
- Other queries can be provided as needed by the module

A module’s query function

- Will be passed four parameters
  - The current model’s instance of the module
  - The name of the buffer being queried
  - The slot being queried
  - The value to check

For the Demo module

- Only handling the required queries
- We will not have any errors by the module
  - "State error" queries will always be nil
- The module needs to indicate it's busy while it is building a chunk for the create buffer
  - Will report as busy regardless of which buffer is queried for simplicity
- I like to provide warnings when bad values come in

```
(defun demo-module-query (demo b slot value)
  (case slot
    (state
     (case value
       (error nil)
       (busy (demo-module-busy demo))
       (free (not (demo-module-busy demo)))
       (t (print-warning
            "Bad state query to ~s buffer"
            b))))
    (t (print-warning
     "Invalid slot ~s in query to buffer ~s"
      query b))))
```

The requests

- Request for some action to be performed by the module
- The module can do whatever it wants in response to the request
- No specific requirements

The module’s request function

- Will be passed three parameters
  - The current model’s instance of the module
  - The name of the buffer to which the request was made
  - A *chunk-spec* describing the request
- The return value of the request function is ignored

What is a chunk-spec?

- A specification of a chunk
  - An internal representation of what one sees in a production request (RHS +*buffer*) or buffer test (LHS =*buffer*)
  - Has the chunk-type name
  - Zero or more test-slot-value triples
    - =, -, <, >, <=, >=
    - Slot name symbol
    - Slot value

The chunk-spec accessors

- The low-level representation of a chunk-spec isn't part of the API
- Specific functions for accessing the components are the API

- Chunk-spec-chunk-type
  - Takes a chunk-spec
  - returns the name of the chunk-type in the chunk-spec
- Slot-in-chunk-spec-p
  - Takes a chunk-spec and a slot name
  - Returns non-nil if that slot is used in the chunk-spec
- Chunk-spec-slot-spec
  - Takes a chunk-spec and an optional slot name
  - Returns a list of slot description lists for the given slot or all slots if none provided
    - Slot description list is a three element list of
      - test symbol
      - Slot name symbol
      - Slot value

Example chunk-spec usage

- Assume we have a chunk-spec for this request from a production bound to *foo*:

```
+visual-location>
 isa visual-location
 >  screen-x 10
 <= screen-x 100
    screen-x lowest
 <  screen-y 20
    kind text
```

```
> (chunk-spec-chunk-type *foo*)
VISUAL-LOCATION

> (slot-in-chunk-spec-p *foo* 'screen-x)
SCREEN-X

> (slot-in-chunk-spec-p *foo* 'whatever)
NIL

> (chunk-spec-slot-spec *foo*)
((> SCREEN-X 10) (<= SCREEN-X 100) (= SCREEN-X LOWEST)
 (< SCREEN-Y 20) (= KIND TEXT))

> (chunk-spec-slot-spec *foo* 'screen-x)
((> SCREEN-X 10) (<= SCREEN-X 100) (= SCREEN-X LOWEST))

> (chunk-spec-slot-spec *foo* 'whatever)
#|Warning: Slot WHATEVER is not specified in the chunk-
    spec. |#
NIL
```

The Request functions

```
(defun demo-module-requests (demo buffer spec)
  (if (eq buffer 'create)
     (demo-create-chunk demo spec)
    (demo-handle-print-out spec)))


(defun demo-handle-print-out (spec)
  (let* ((type (chunk-spec-chunk-type spec))
         (value? (slot-in-chunk-spec-p spec 'value))
         (v1 (when value? (chunk-spec-slot-spec spec 'value))))
     (if (eq type 'demo-output)
         (if value?
             (if (= (length v1) 1)
                 (if (eq (caar v1) '=)
                     (model-output "Value: ~s" (caddar v1))
                   (model-warning "Invalid slot modifier ~s" (caar v1)))
               (model-warning "Value slot specified multiple times"))
           (model-warning "Value slot missing in output buffer request"))
       (model-warning "bad chunk-type in request to output buffer"))))
```

Another chunk-spec command

- Chunk-spec-to-chunk-def
  - Takes a chunk-spec
  - If that chunk-spec only uses the = test on the slots and each slot is specified at most once it returns a list that is valid for passing to define-chunks to create a chunk
- Assume we have a chunk-spec for this request bound to *foo*:

```
+goal>
  isa visual-location
  screen-x 10
  screen-y 20
```

```
> (chunk-spec-to-chunk-def *foo*)
(ISA VISUAL-LOCATION SCREEN-X 10 SCREEN-Y 20)
> (define-chunks-fct (list (chunk-spec-to-chunk-def *foo*)))
(VISUAL-LOCATION0)
```

```
> (pprint-chunks visual-location0)
VISUAL-LOCATION0
  ISA VISUAL-LOCATION
   SCREEN-X  10
   SCREEN-Y  20
   DISTANCE  NIL
   KIND  NIL
   COLOR  NIL
   VALUE  NIL
   HEIGHT  NIL
   WIDTH  NIL
   SIZE  NIL
```

Demo-create-chunk

```
(defun demo-create-chunk (demo spec)
   (if (demo-module-busy demo)
      (model-warning "Cannot handle request when busy")
     (let* ((chunk-def (chunk-spec-to-chunk-def spec))
            (c (when chunk-def
                 (car (define-chunks-fct (list chunk-def))))))
        (when c
          (let ((delay (if (demo-module-esc demo)
                           (demo-module-delay demo) 0)))
            (setf (demo-module-busy demo) t)
           ;; put the chunk into the buffer
           ;; and set the module back to free
           ;; after delay seconds have passed
           )))))
```

Creating events

- Whenever a module needs to do things at a particular time it needs to generate events
  - Specify when the event occurs
  - Specify what to do at that time
- The events go on the queue and get executed at the appropriate time
- They are also printed in the model's trace

Event generation

- Lots of functions for doing so
  - See the manual for details
- We'll look at two in particular
- A very specific one
  - schedule-set-buffer-chunk
- A general one
  - schedule-event-relative

Schedule-set-buffer-chunk

```
(defun schedule-set-buffer-chunk
       (buffer-name chunk-name time-delta &key (module :none) …)
 …)
```

- Give it
  - name of a buffer
  - name of the chunk
  - how far ahead in seconds to do the setting
  - a module name for the trace
- Will generate the event to perform that action
- Shows up like this in the trace

```
0.200   DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL A
```

Schedule-event-relative

```
(defun schedule-event-relative (time-delay action
                                &key (params nil) (module :none) …)
  …)
```

- Give it
  - How far ahead in seconds to do the action
  - A function to call at that time
  - The list of parameters to pass the function
  - A module name for the trace
- Will generate the event to perform that action
- At the specified time the function will be called with those parameters

Shifter-create-chunk

```
(defun demo-create-chunk (demo spec)
   (if (demo-module-busy demo)
      (model-warning "Cannot handle request when busy")
     (let* ((chunk-def (chunk-spec-to-chunk-def spec))
            (c (when chunk-def
                (car (define-chunks-fct (list chunk-def))))))
        (when c
          (let ((delay (if (demo-module-esc demo)
                           (demo-module-delay demo) 0)))
            (setf (demo-module-busy demo) t)
            (schedule-set-buffer-chunk 'create c delay
                                       :module 'demo)
            (schedule-event-relative delay 'free-demo-module
                          :params (list demo) :module 'demo)))))))
```

Finished

```
(defun free-demo-module (demo)
  (setf (demo-module-busy demo) nil))

(define-module-fct 'demo  '(create output)
  (list (define-parameter :create-delay
               :documentation "time to create the chunk for the demo module"
               :default-value .1
               :valid-test (lambda (x) (and (numberp x) (>= x 0)))
               :warning "Non-negative number" :owner t)
        (define-parameter :esc :owner nil))

   :version "1.0a1"
   :documentation "Demo module for ICCM tutorial"
   :creation 'create-demo-module
   :reset 'reset-demo-module
   :delete 'delete-demo-module
   :params 'demo-module-params
   :request 'demo-module-requests
   :query 'demo-module-queries
)
```

Module complete

- That covers the basics
- Other things that can be done
  - See the existing modules and manuals for examples and details
- Traced all the module functions and ran a simple model
  - Next few slides show when things are getting called

Running a simple model

```
(clear-all)
(define-model test-demo-module
   (sgp :esc t :create-delay .15)

 (p p1                          (p p2
      ?create>                      =create>
      state free                    isa visual-location
      buffer empty                  ==>
      ==>                           +output>
      +create>                      isa demo-output
      isa visual-location           value =create
      screen-x 10                   ))
      screen-y 20)
```

The trace (load time)

```
; Loading C:\Documents and Settings\Root\Desktop\demo-model.lisp
 0[4]: (CREATE-DEMO-MODULE TEST-DEMO-MODULE)
 0[4]: returned #S(DEMO-MODULE :DELAY NIL :BUSY NIL :ESC NIL)
 0[4]: (RESET-DEMO-MODULE #S(DEMO-MODULE :DELAY NIL :BUSY NIL :ESC NIL))
 0[4]: returned DEMO-OUTPUT
 0[4]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY NIL :BUSY NIL :ESC NIL)
   (:CREATE-DELAY . 0.1))
 0[4]: returned 0.1
 0[4]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :BUSY NIL :ESC NIL)
   (:ESC))
 0[4]: returned NIL
 0[4]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :BUSY NIL :ESC NIL)
   (:ESC . T))
 0[4]: returned T
 0[4]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :BUSY NIL :ESC T)
   (:CREATE-DELAY . 0.15))
 0[4]: returned 0.15
```

The trace (run time)

```
> (run .25)
0.000   PROCEDURAL              CONFLICT-RESOLUTION
 0[4]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL)
                            CREATE STATE FREE)
 0[4]: returned T
0.000   PROCEDURAL              PRODUCTION-SELECTED P1
0.000   PROCEDURAL              QUERY-BUFFER-ACTION CREATE
0.050   PROCEDURAL              PRODUCTION-FIRED P1
0.050   PROCEDURAL              MODULE-REQUEST CREATE
 0[4]: (DEMO-MODULE-REQUESTS #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL)
                             CREATE    #S(ACT-R-CHUNK-SPEC …))
 0[4]: returned #S(ACT-R-EVENT …)
0.050   PROCEDURAL              CLEAR-BUFFER CREATE
0.050   PROCEDURAL              CONFLICT-RESOLUTION
 0[4]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY T)
                              CREATE STATE FREE)
 0[4]: returned NIL
0.200   DEMO                    SET-BUFFER-CHUNK CREATE VISUAL-LOCATION0
0.200   DEMO                    FREE-DEMO-MODULE #S(DEMO-MODULE :DELAY
   0.15 :ESC T :BUSY T)
```

```
0.200   PROCEDURAL              CONFLICT-RESOLUTION
0.200   PROCEDURAL              PRODUCTION-SELECTED P2
0.200   PROCEDURAL              BUFFER-READ-ACTION CREATE
0.250   PROCEDURAL              PRODUCTION-FIRED P2
0.250   PROCEDURAL              MODULE-REQUEST OUTPUT
 0[4]: (DEMO-MODULE-REQUESTS #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL)
                             OUTPUT #S(ACT-R-CHUNK-SPEC …))
Value: VISUAL-LOCATION0-0
 0[4]: returned NIL
0.250   PROCEDURAL              CLEAR-BUFFER CREATE
0.250   PROCEDURAL              CLEAR-BUFFER OUTPUT
0.250   PROCEDURAL              CONFLICT-RESOLUTION
 0[4]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL)
                            CREATE STATE FREE)
 0[4]: returned T
0.250   PROCEDURAL              PRODUCTION-SELECTED P1
0.250   PROCEDURAL              QUERY-BUFFER-ACTION CREATE
0.250   ------                  Stopped because time limit reached
CG-USER(31): (clear-all)
 0[4]: (DELETE-DEMO-MODULE #S(DEMO-MODULE :DELAY 0.15 :BUSY NIL :ESC T))
 0[4]: returned NIL
NIL
```

BOLD Predictions for Every Model

John R. Anderson

Goals of Enterprise

- ACT-R’s “sweet spot” is understanding how the various pieces of cognition get brought together to perform complex mental functions.
- This is the unique essence of the human mind.
- Since we are not just interested in producing high functionality behavior, we want to know whether ACT-R and our models describe what is happening in the human head.
- Frequently behavioral data is impoverished relatively to the complexity of the models we are producing.
- Therefore, we would like some converging data that tests the activity of the modules.
- As a bonus we anchor ACT-R in the brain.
- This can guide development of models.
- This can guide development of the architecture.
- This can help organize brain imaging data.

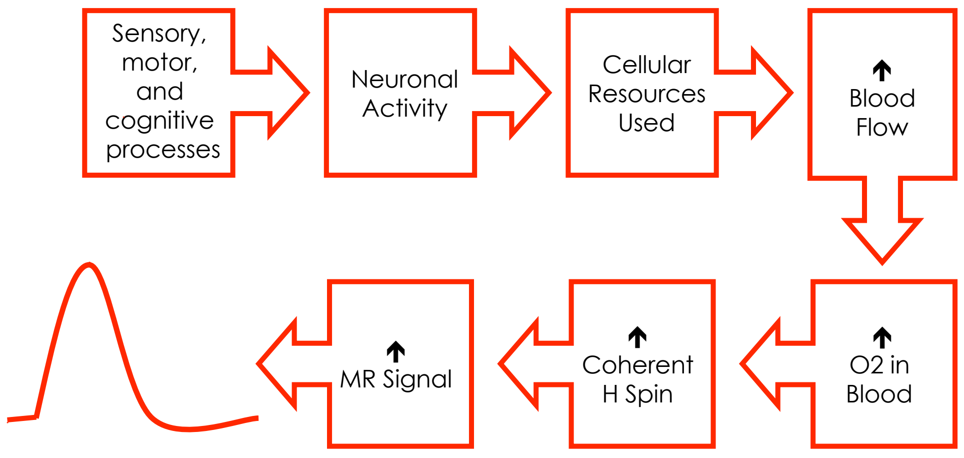
Functional MRI (fMRI)

- What does it mean to take an IMAGE of FUNCTION?
- What we want is a representation of BRAIN ACTIVITY over space, so that we can distinguish ACTIVE regions from INACTIVE regions
- The brain doesn't really glow when it's active (cf. optical imaging)
- So what are we imaging in fMRI?

BOLD Contrast

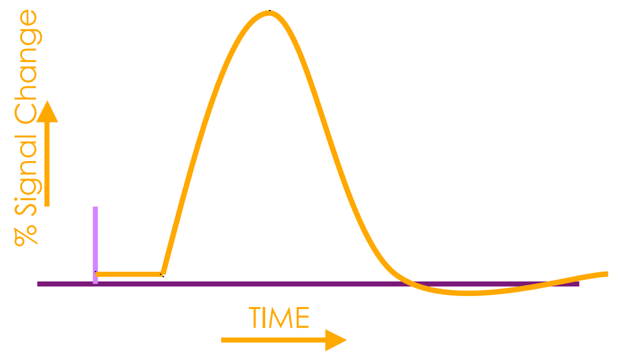
- Blood-Oxygen-Level Dependent fMRI
- When an area of the brain is active, blood is sent there
- This blood is highly OXYGENATED from the lungs and displaces DEOXYGENATED blood in the active region
- The net result is that brain activity leads to increased oxygen in the blood
- Oxygenated and deoxygenated blood are magnetically different and this difference can be detected using MRI

What are we measuring?

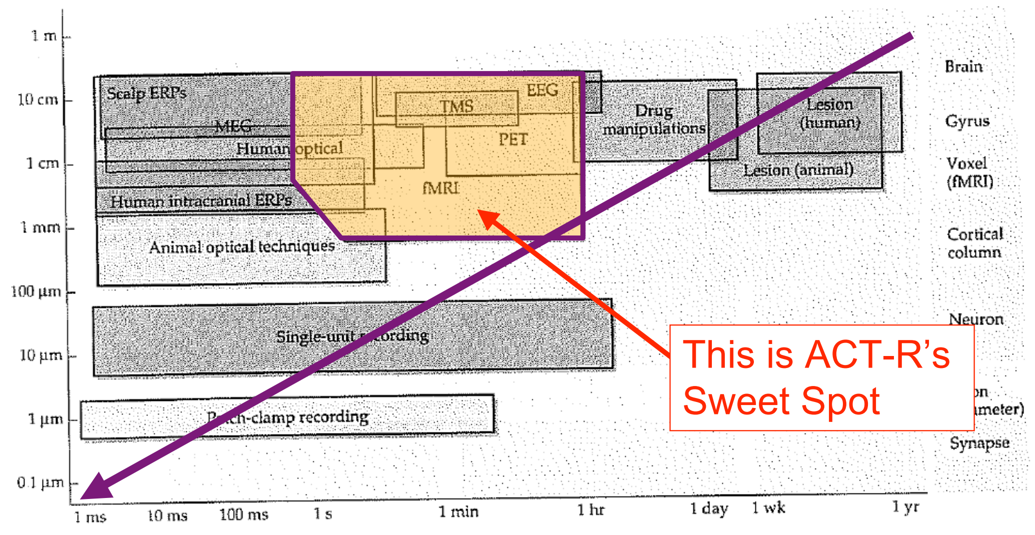


BOLD Response

- The MR signal response to neural activity has a distinctive shape that is slow (20 s) and delayed

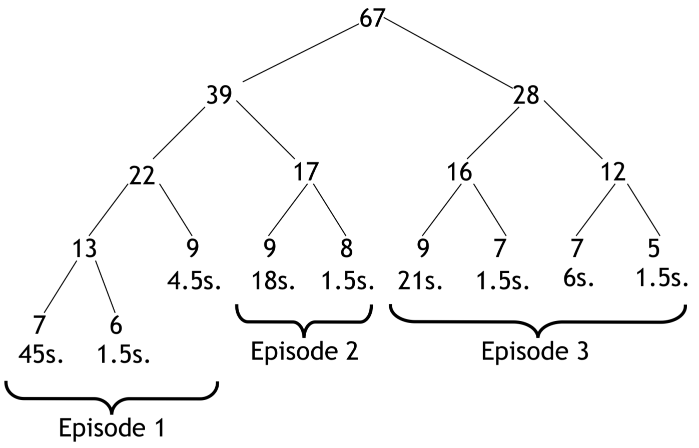


Spatial and Temporal Resolution



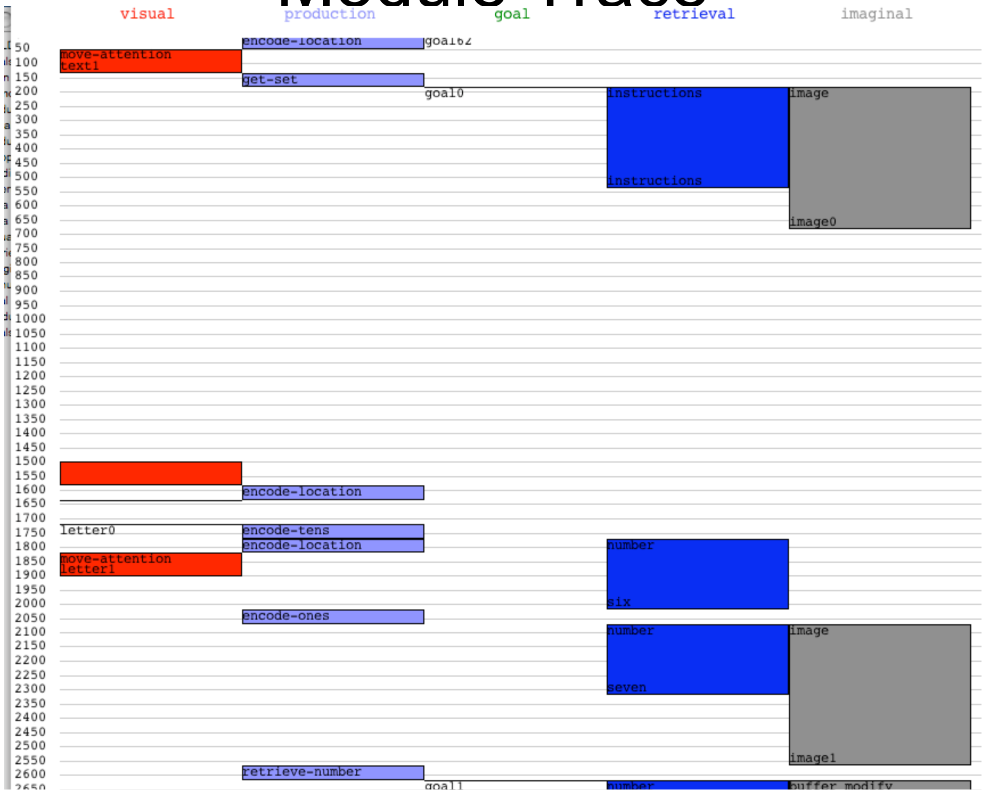
Yulin Qin's Mental Arithmetic Task

Recursively divide a number n into a sum of a and b as follows:
(1) Find *a*. This is calculated as half of the number *n* (rounding down if necessary) plus the tens digit (e.g., in the case of 67 the number a is 33 + 6 = 39).
(2) Calculate *b* as *n* – *a* (in the example, 67 - 39 = 28).
(3) Decompose *a* first and then *b* (storing *b* as a subgoal to be retrieved later).
(4) When the decomposition reaches a single-digit number, type it.

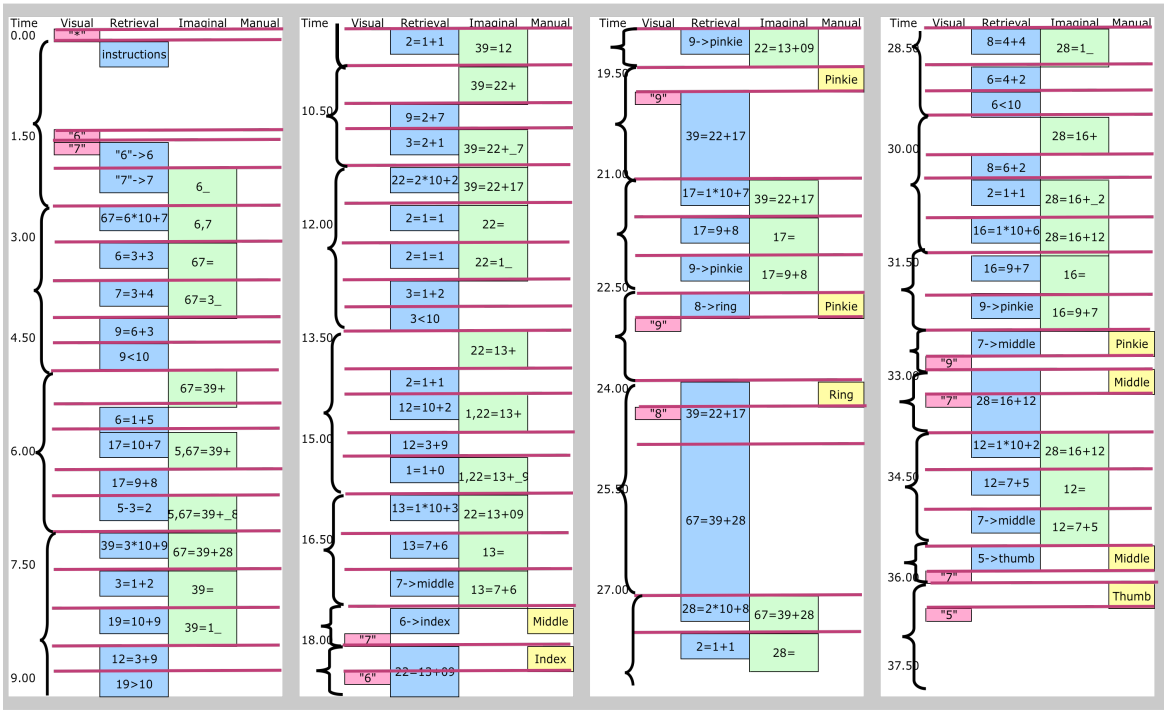


Demo

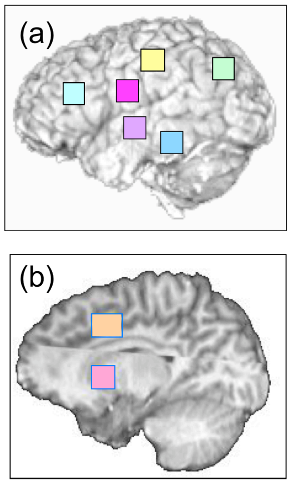
Module Trace



“Ideal” Module Trace

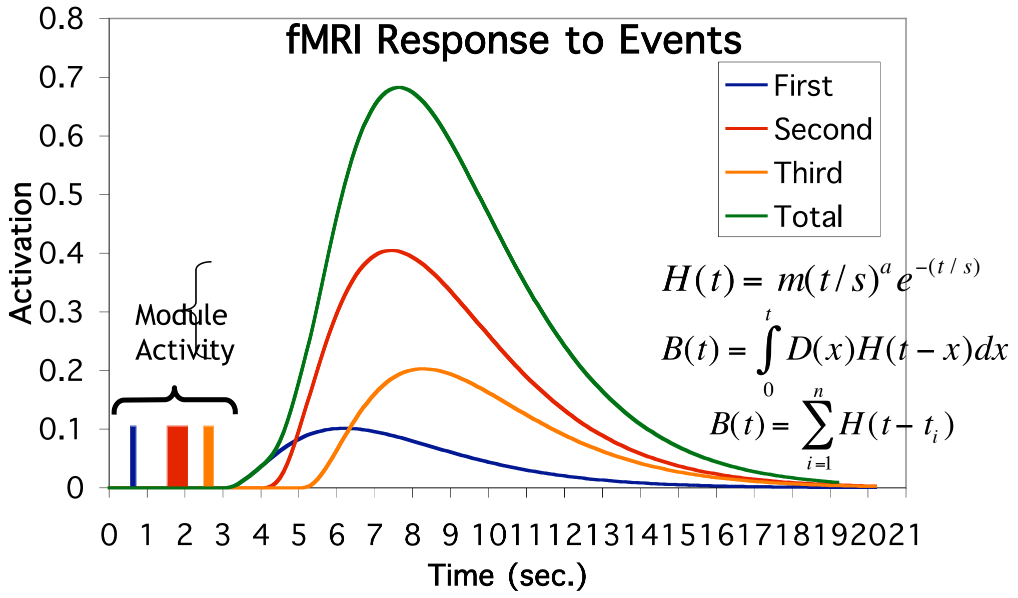


Module-Brain Mappings



		x	y	z	Region
Manual	🟨	37	-25	47	Motor
Imaginal	🟩	23	-64	34	Parietal
Vocal	🟪	44	-12	29	Motor
Declarative	🟦	40	21	21	Prefrontal
Aural	🟪	47	-16	4	Auditory
Visual	🟦	42	-60	-8	Fusiform
Goal	🟧	5	10	38	ACC
Procedural	🟥	15	9	2	Caudate

Mapping Module Activity onto the BOLD Response



$$H(t) = m(t/s)^a e^{-(t/s)}$$

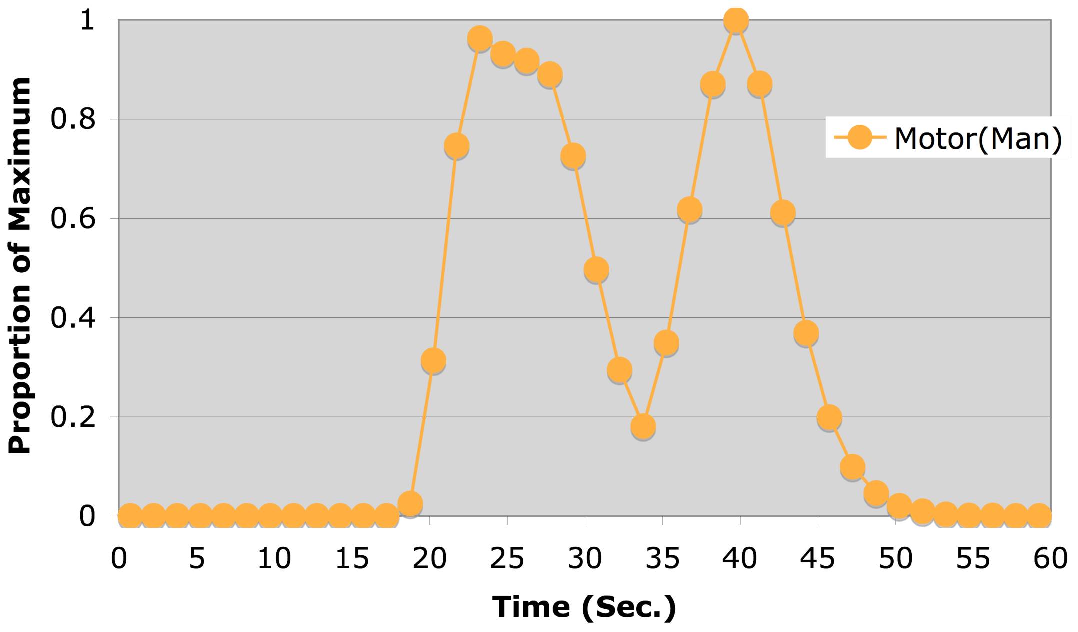
$$B(t) = \int_0^t D(x)H(t-x)dx$$

$$B(t) = \sum_{i=1}^{n} H(t-t_i)$$

Demo

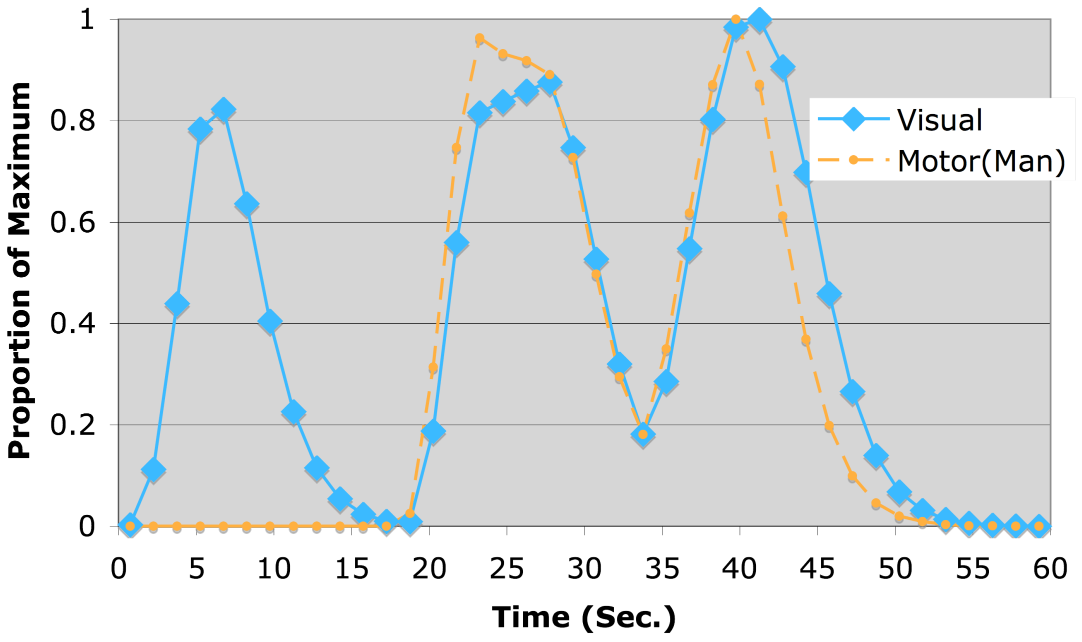
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed



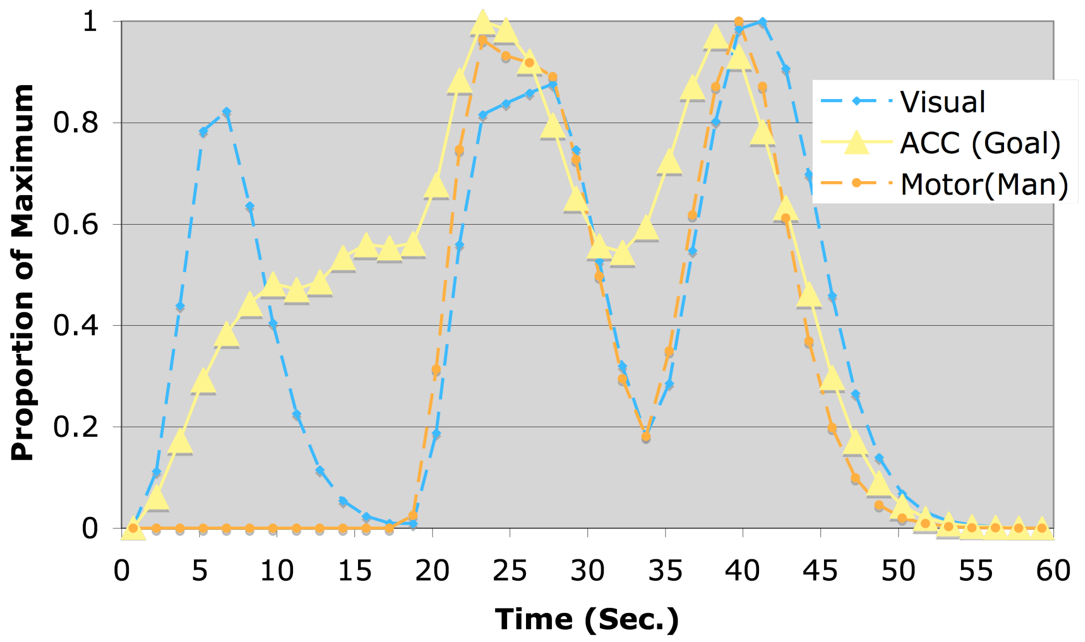
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed



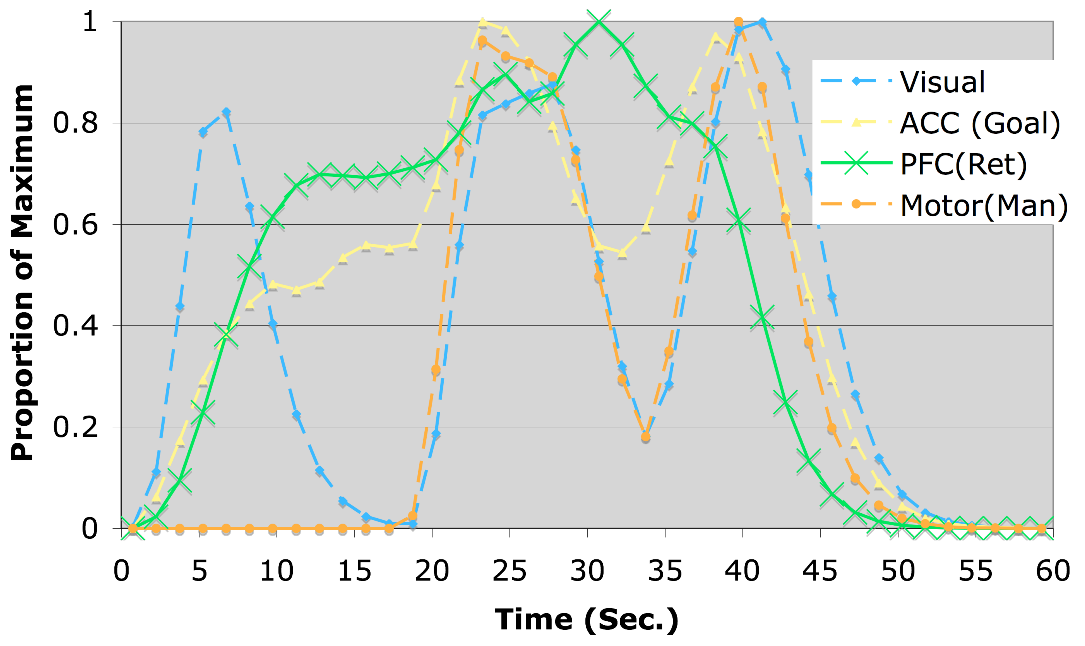
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed



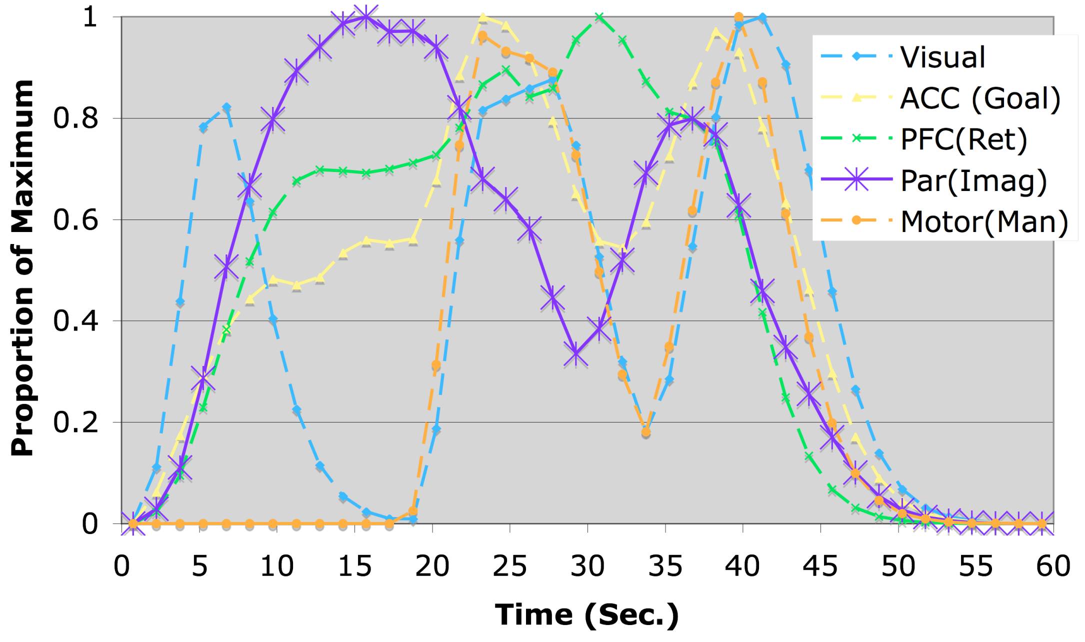
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed



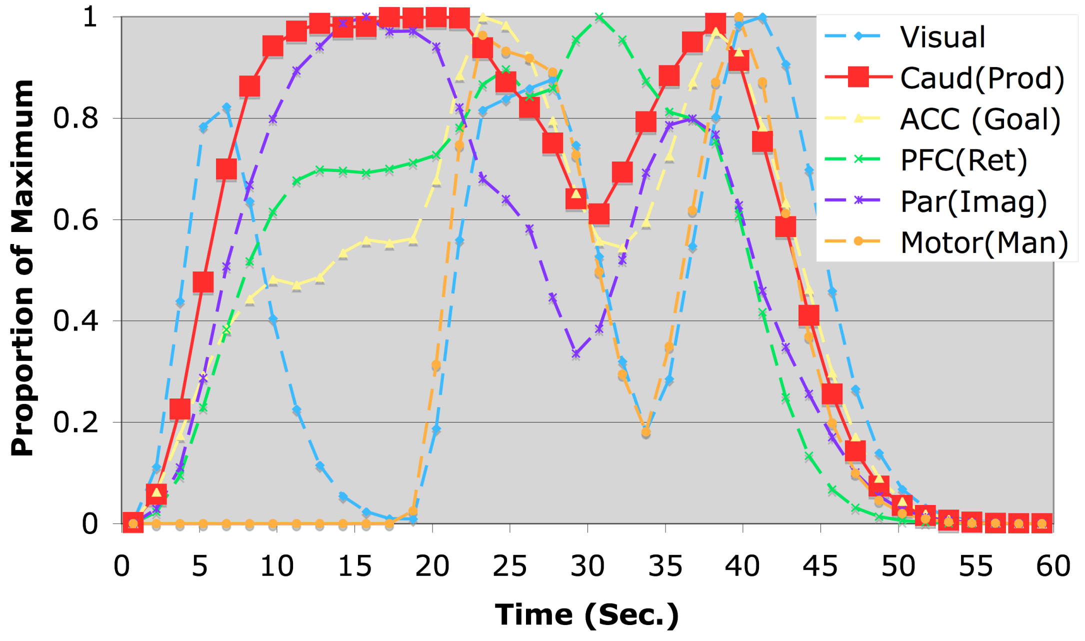
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed

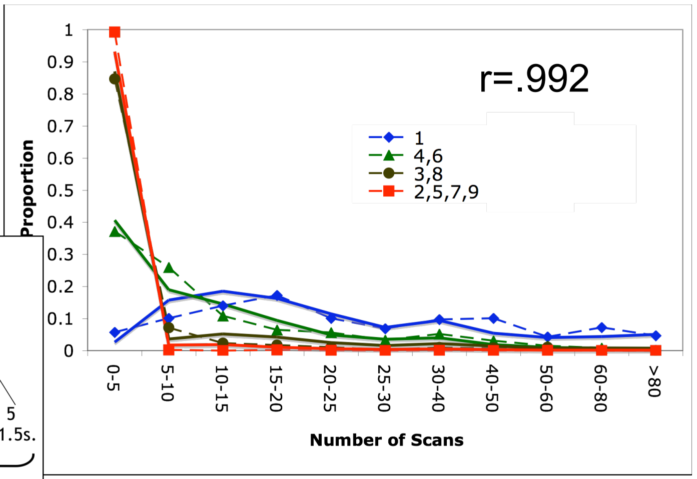


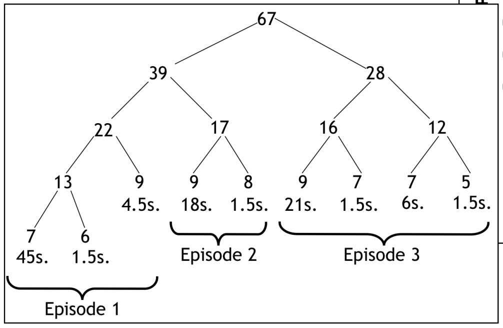
Predictions from Run (1.5 sec scans)

$H(t) = m(t/s)^a e^{-(t/s)}$ a = 4; s = 1.2; max a*s = 4.8 sec delayed



Latency Distributions



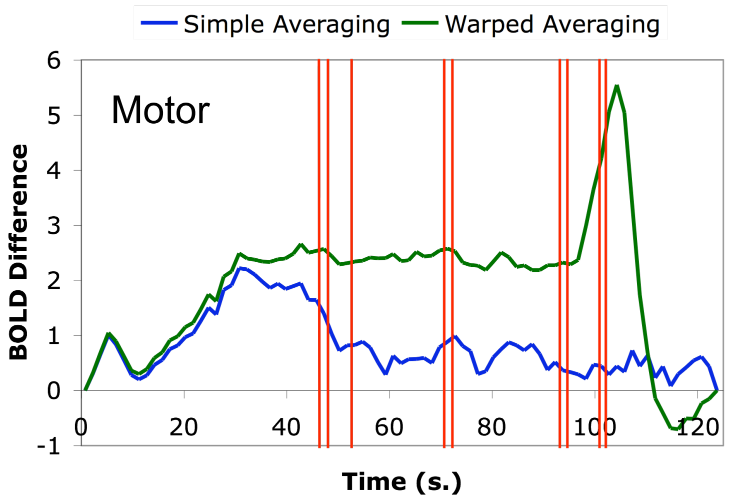


➢Experiment involve 10 different digits (59, 61, 62, 63, 64, 65, 66, 67, 70, 71) with longer digits tending to take longer.
➢Model randomly varied latency factor from 0 to 1.5 sec. (1 in previous).
➢Retrieval threshold was -.5 and activation noise .25 (-10 and 0 in previous).
➢Model randomly varied imaginal updates from 0 to 1.5 .sec (.5 in previous).
➢Model had imaginal failures according to a Poisson process with mean time of 20 sec. (infinite in previous)
➢Ran 1000 runs for each of 10 digits and compared with ~250 observations.

Data Warping: The Problem

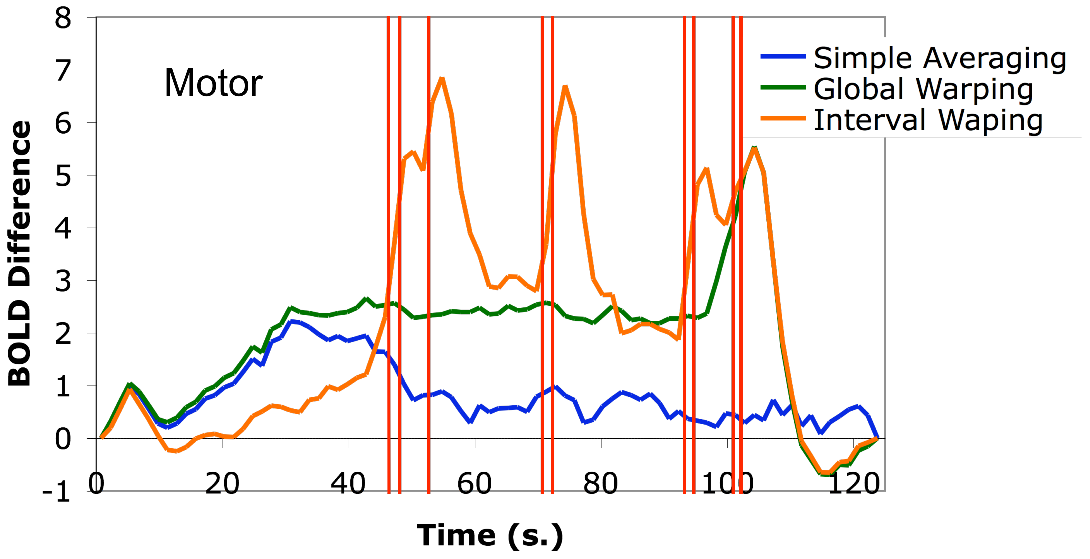
➢How do we present data when trial length varies by a factor of 10 or more?
➢Any scheme of just averaging trials together would total blur temporal structure.
➢Warp all scans to average length using "Split-Fincham" procedure that grows from ends.

To warp a scan of length n to mean length m:
➢If n > m, take m/2 scans from the beginning and m/2 from the end.
➢If n < m, take n/2 from beginning and pad with last & take n/2 from end and pad with first

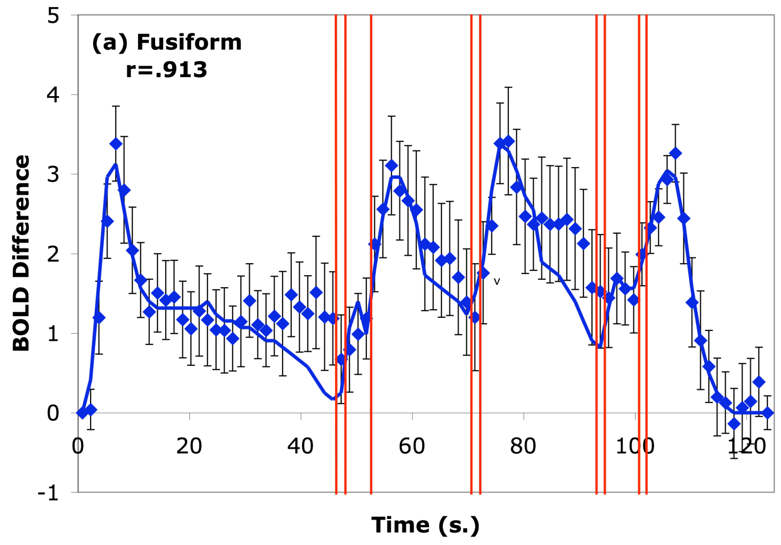


Data Warping: The Solution

- ➢Separately warp the scans for each interval defined by observable events -- stimulus presentation and responses.
- ➢This creates scan sequences that preserve the temporal structure of the beginning and end of the intervals and just represent the approximate average activity in their middle.



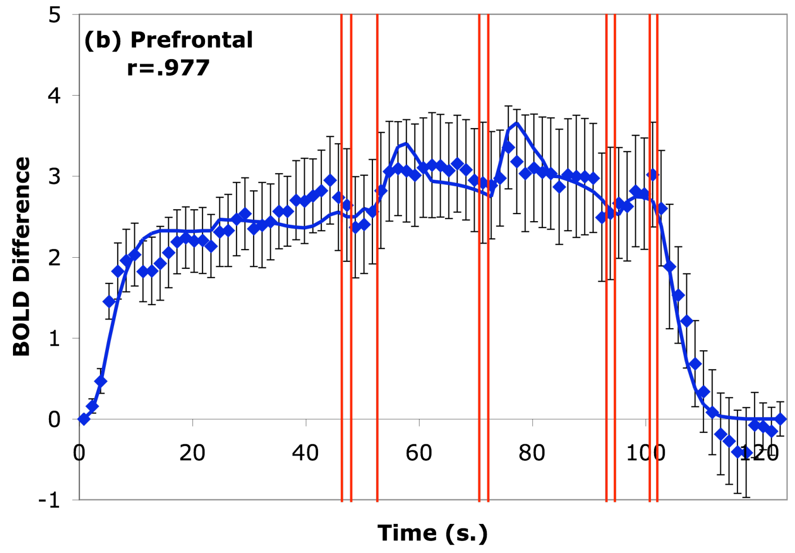
Visual Predicts Fusiform



- ➢Apply same warping procedure to model & subjects
- ➢Standard error of points calculated across subjects.
- ➢Driven by encoding of digits
- ➢Only a single magnitude parameter is estimated for the region

$$\chi^2_{80} = \frac{\sum_{i=1}^{81} (\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 38.32$$

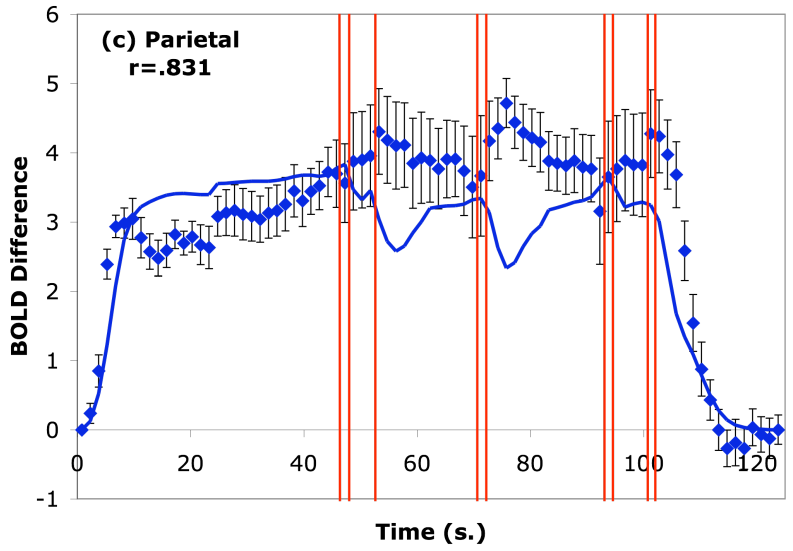
Retrieval Predicts Prefrontal



➢Note rises where past results are being retrieved

$$\chi^2_{80} = \frac{\sum_{i=1}^{81}(\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 21.17$$

Imaginal Does Not Predict Parietal

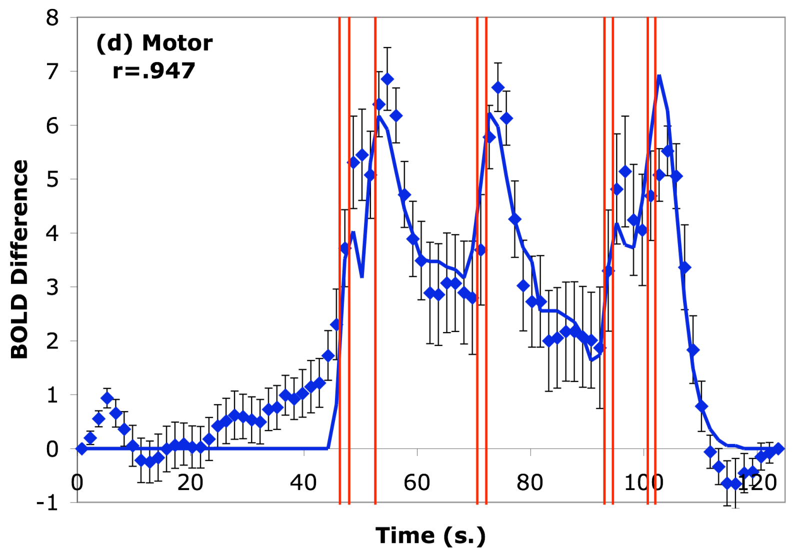


➢Imaginal fails to fall off at points where past results are being retrieved.

➢Can be fit as a combination of imaginal, visual, and manual.

$$\chi^2_{80} = \frac{\sum_{i=1}^{81}(\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 382.14$$

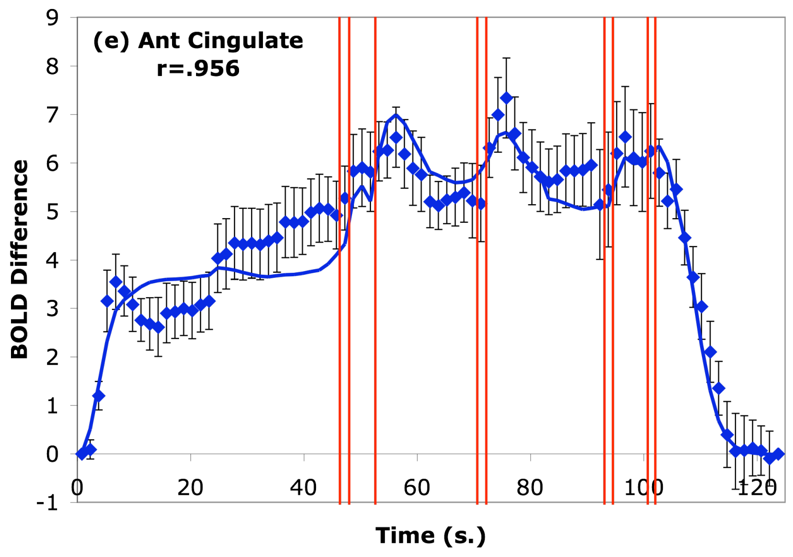
Manual Almost Predicts Motor



➢Missing early preparatory motor actions.

$$\chi^2_{80} = \frac{\sum_{i=1}^{81}(\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 186.41$$

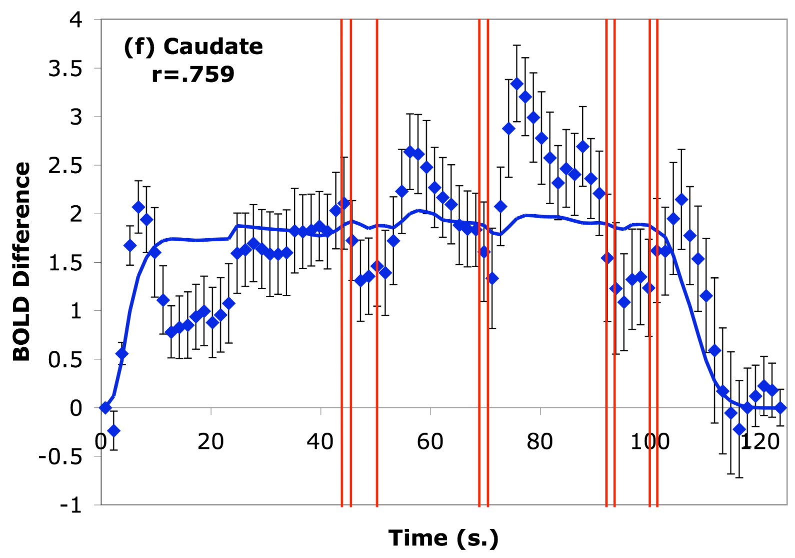
Goal Predicts Anterior Cingulate Cortex



➢Note sweeps up as goals switch rapidly during motor output

$$\chi^2_{80} = \frac{\sum_{i=1}^{81}(\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 71.40$$

Productions do not Predict Caudate



➢Can be fit by adding a sensitivity to visual onsets

$$\chi^2_{80} = \frac{\sum_{i=1}^{81} (\hat{B}_i - \overline{B}_i)^2}{s^2_{\overline{B}}} = 155.71$$

Goals of Enterprise

➢Since we are not just interested in producing high functionality behavior, we want to know whether ACT-R and our models describe what is happening in the human head.
➢Frequently behavioral data is impoverished relatively to the complexity of the models we are producing.
➢Imaging data tests the activity of the modules and anchors ACT-R in the brain.
➢This can guide development of models -- preparatory motor activity.
➢This can guide development of the architecture -- parietal data imply imaginal module reflects both external and internal representational operations.
➢This can help organize brain imaging data -- caudate reflects both procedural execution and visual processing.

ACT-R/Lisa

Learning from Instructions
and Skill Acquisition

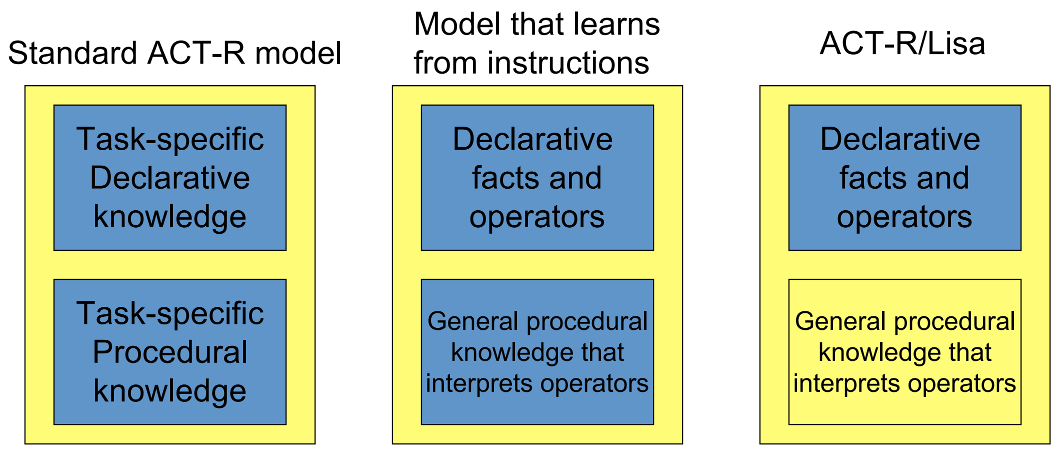
Niels Taatgen

Goals

- Unify several models of learning from instructions (ATC, Algebra, CMU-ASP, FMS, etc.)
- Make modeling easier by specifying a task at a higher level (ACT-Simple on steroids)
- Make existing modeling paradigms (e.g., instance retrieval) easily available
- Abstract away from nitty-gritty ACT-R/PM

Contents

- Theoretical background
- Example model (counting)
- How is ACT-R/Lisa implemented
- What is the syntax of an operator
- Some more elaborate examples



Standard ACT-R model
Task-specific Declarative knowledge
Task-specific Procedural knowledge
Model that learns from instructions
Declarative facts and operators
General procedural knowledge that interprets operators
ACT-R/Lisa
Declarative facts and operators
General procedural knowledge that interprets operators

ACT-R/Lisa provides:

A standard set of production rules to interpret operators

Some syntax to make operators more compact and readable

Caveat

- For most models the level of detail of the instructions will be higher than actual instructions subjects get
- “Real” instructions assume much more background skills than ACT-R typically has

```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :state start :action (read from) :new-state step1)
(op read-to :state step1 :action (read to) :new-state step2)
(op say-current :state step2 :action (say from) :new-state step3)
(op advance-count :state step3 :pre (- from to) :action (retrieve count-
   fact from) :new-state step2)
(op count-done :state step3 :pre (from to) :action (repeat)))
```

```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :state start :action (read from) :new-state step1)
(op read-to :state step1 :action (read to) :new-state step2)
(op say-current :state step2 :action (say from) :new-state step3)
(op advance-count :state step3 :pre (- from to) :action (retrieve count-
   fact from) :new-state step2)
(op count-done :state step3 :pre (from to) :action (repeat)))
```

```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :state start :action (read from) :new-state step1)
(op read-to :state step1 :action (read to) :new-state step2)
(op say-current :state step2 :action (say from) :new-state step3)
(op advance-count :state step3 :pre (- from to) :action (retrieve count-
   fact from) :new-state step2)
(op count-done :state step3 :pre (from to) :action (repeat)))
```

```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :state start :action (read from) :new-state step1)
(op read-to :state step1 :action (read to) :new-state step2)
(op say-current :state step2 :action (say from) :new-state step3)
(op advance-count :state step3 :pre (- from to) :action (retrieve count-
   fact from) :new-state step2)
(op count-done :state step3 :pre (from to) :action (repeat)))
```

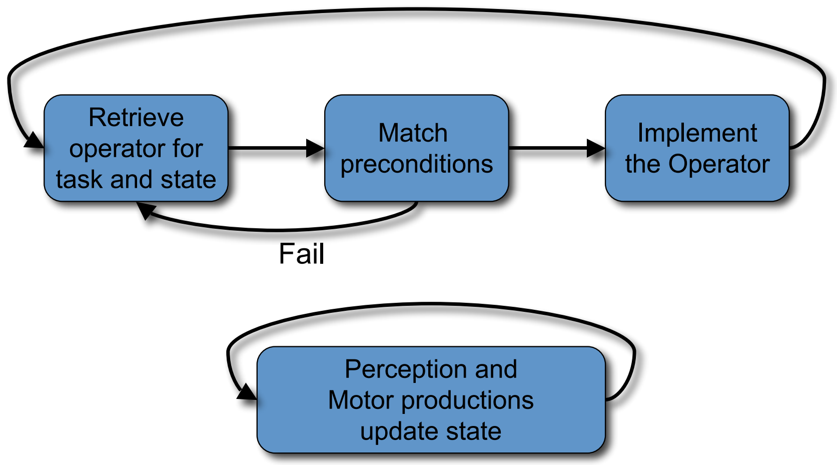
```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :action (read from) )
(op read-to :pre from :action (read to))
(op say-current :pre from :pre (- vocal) :action (say from))
(op advance-count :pre (- from to) :pre (vocal spoken) :action (retrieve
   count-fact from))
(op count-done :pre (from to) :pre (vocal spoken) :action (repeat))
(order-operators read-from read-to say-current advance-count say-current)
```

```
(define-model-lisa count count (from to)

;;; Counting model facts

(add-fact order0 count-fact "0" "1")
(add-fact order1 count-fact "1" "2")
(add-fact order2 count-fact "2" "3")
etc.
;;; counting model operators

(op read-from :pre (- from) :action (read from) )
(op read-to :pre from :action (read to))
(op say-current :pre from :pre (- vocal) :action (say from))
(op advance-count :pre (- from to) :pre (vocal spoken) :action (retrieve
   count-fact from))
(op count-done :pre (from to) :pre (vocal spoken) :action (repeat))
(order-operators read-from read-to say-current advance-count say-current)
```

Running the model

- Load in ACT-R
- Load in Lisa
- Load in your Lisa model and run it as you would run an ACT-R model

Operator cycle


Retrieve operator for task and state
Match preconditions
Implement the Operator
Fail
Perception and Motor productions update state

```
0.15   Operator READ-FROM  with action READ(FROM) was selected.
0.78    Action: Reading "3" into variable FROM
0.98   Operator READ-TO  with action READ(TO) was selected.
1.04    Action: Reading "5" into variable TO
1.24   Operator SAY-CURRENT  with action SAY(FROM) was selected.
1.29    Action: Saying "3"
1.64   Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM) was selected.
1.82    Action: Retrieving a COUNT-FACT fact with cue "3", storing result "4" in FROM
2.12   Operator SAY-CURRENT  with action SAY(FROM) was selected.
2.17    Action: Saying "4"
2.47   Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM) was selected.
2.67    Action: Retrieving a COUNT-FACT fact with cue "4", storing result "5" in FROM
2.87   Operator SAY-CURRENT  with action SAY(FROM) was selected.
2.92    Action: Saying "5"
3.22        Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM) was rejected
because it violated preconditions.
3.32        Operator SAY-CURRENT  with action SAY(FROM) was rejected because it violated
preconditions.
3.43   Operator COUNT-DONE  with action REPEAT was selected.
3.48    Action: Repeating the goal
```

After learning:

```
488.64   Operator READ-FROM  with action READ(FROM) was selected.
488.69    Action: Reading "3" into variable FROM
488.69  Compiled Operator SAY-CURRENT  with action SAY(FROM) was selected.
488.69    Action: Saying "3"
488.74  Compiled Operator READ-TO  with action READ(TO) was selected.
488.79    Action: Reading "5" into variable TO
488.94  Compiled Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM)
was selected.
488.94    Action: Retrieving a COUNT-FACT fact with cue "3", storing result "4" in FROM
488.99  Compiled Operator SAY-CURRENT  with action SAY(FROM) was selected.
488.99    Action: Saying "4"
489.24   Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM) was
selected.
489.45    Action: Retrieving a COUNT-FACT fact with cue "4", storing result "5" in FROM
489.50  Compiled Operator SAY-CURRENT  with action SAY(FROM) was selected.
489.50    Action: Saying "5"
489.60        Operator ADVANCE-COUNT  with action RETRIEVE(COUNT-FACT, FROM) was
rejected because it violated preconditions.
489.75  Compiled Operator COUNT-DONE  with action REPEAT was selected.
489.80    Action: Repeating the goal
```

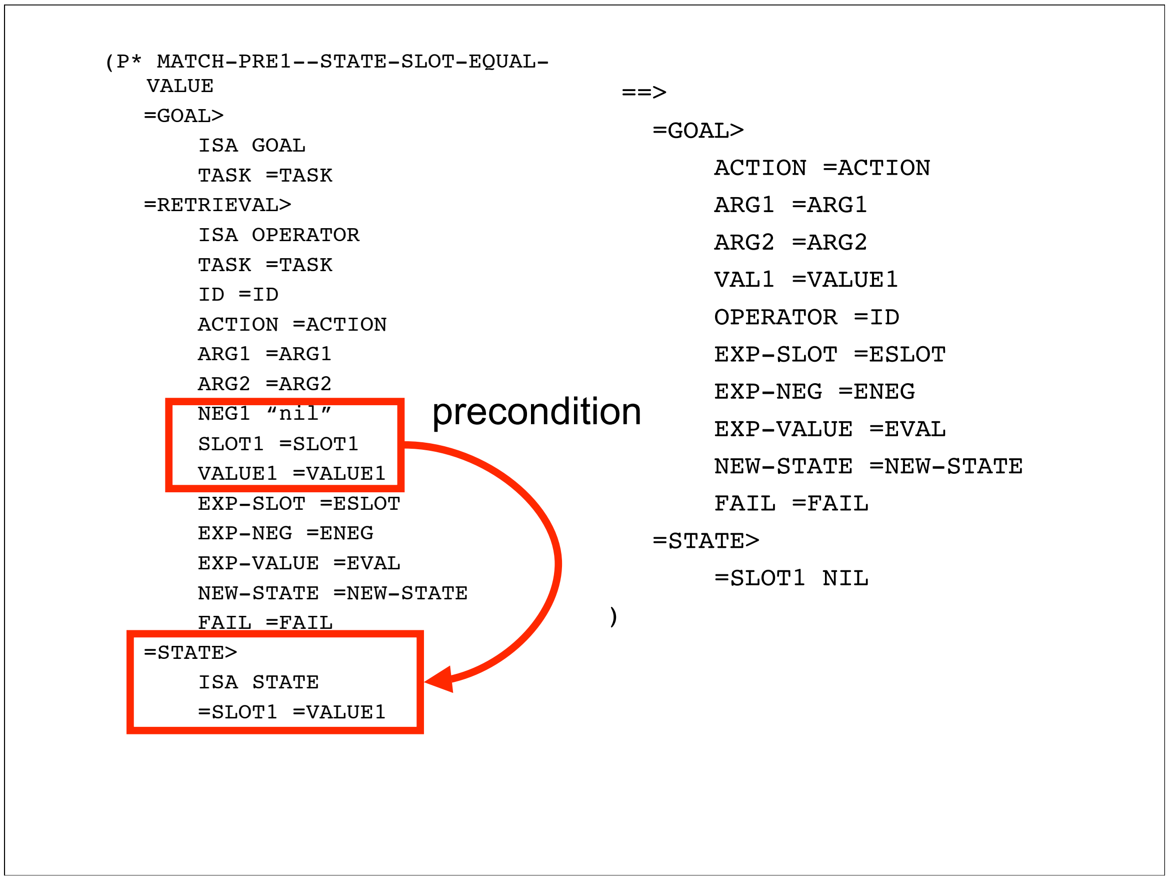
The details of ACT-R/Lisa

- Two new buffers:
  - State: is used to summarize all perceptual and motor buffers (largely for ease of implementation)
  - Var: is used to store the variable bindings and is very similar in use to the imaginal buffer
- The main difference between *state* and *var* is that a slot in the *state* buffer is cleared after it has been read by an operator, but the *var* buffer persists for the duration of a goal

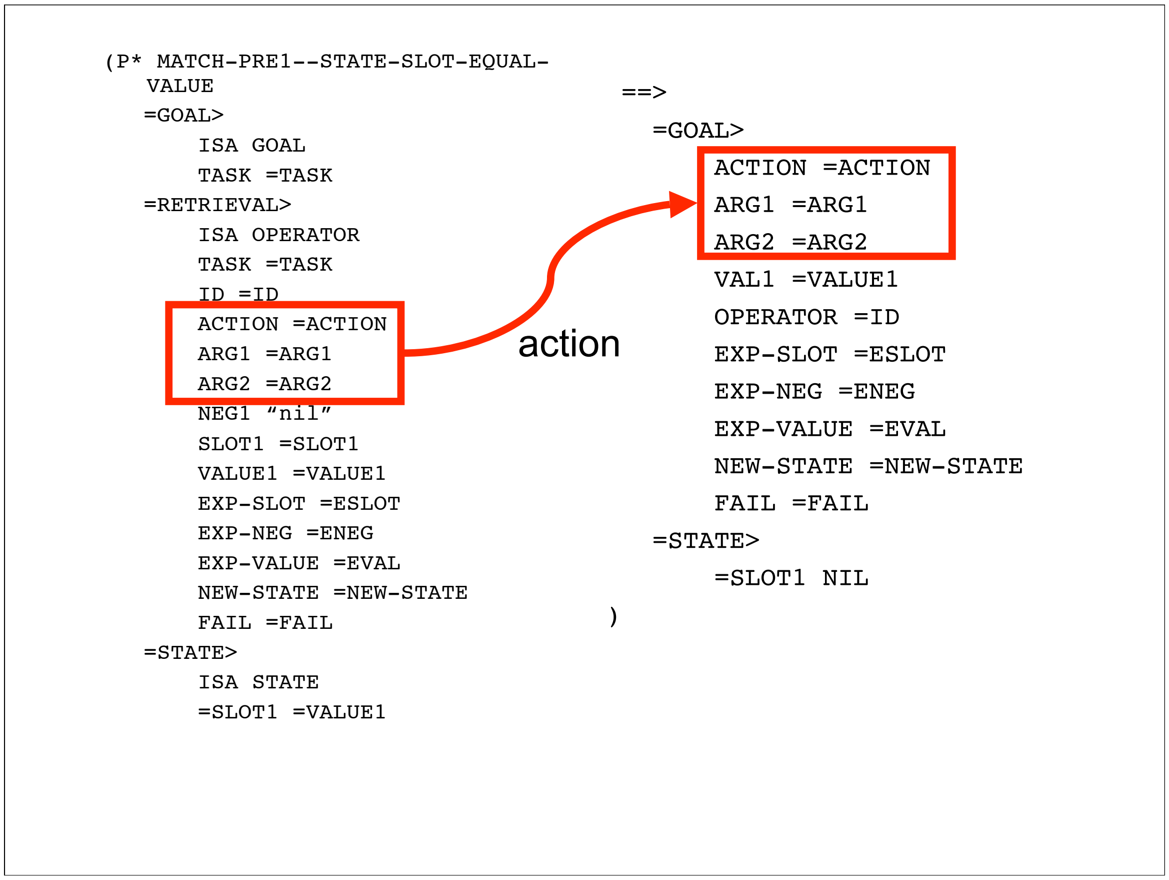
Implementation details

```
(p retrieve-operator
   =goal>
     isa goal
     task =task
     state =s
     action nil
   ?retrieval>
     state free
     buffer empty
==>
   +retrieval>
     isa operator
     task =task
     state =s
     :recently-retrieved nil)
```

```
(P* MATCH-PRE1--STATE-SLOT-EQUAL-
   VALUE
   =GOAL>
       ISA GOAL
       TASK =TASK
   =RETRIEVAL>
       ISA OPERATOR
       TASK =TASK
       ID =ID
       ACTION =ACTION
       ARG1 =ARG1              action
       ARG2 =ARG2
       NEG1 "nil"
       SLOT1 =SLOT1            precondition
       VALUE1 =VALUE1
       EXP-SLOT =ESLOT
       EXP-NEG =ENEG
       EXP-VALUE =EVAL
       NEW-STATE =NEW-STATE
       FAIL =FAIL
   =STATE>
       ISA STATE
       =SLOT1 =VALUE1
==>
  =GOAL>
      ACTION =ACTION
      ARG1 =ARG1
      ARG2 =ARG2
      VAL1 =VALUE1
      OPERATOR =ID
      EXP-SLOT =ESLOT
      EXP-NEG =ENEG
      EXP-VALUE =EVAL
      NEW-STATE =NEW-STATE
      FAIL =FAIL
  =STATE>
      =SLOT1 NIL
)
```



```
(P* MATCH-PRE1--STATE-SLOT-EQUAL-
   VALUE
   =GOAL>
       ISA GOAL
       TASK =TASK
   =RETRIEVAL>
       ISA OPERATOR
       TASK =TASK
       ID =ID
       ACTION =ACTION
       ARG1 =ARG1
       ARG2 =ARG2
       NEG1 "nil"
       SLOT1 =SLOT1
       VALUE1 =VALUE1
       EXP-SLOT =ESLOT
       EXP-NEG =ENEG
       EXP-VALUE =EVAL
       NEW-STATE =NEW-STATE
       FAIL =FAIL
   =STATE>
       ISA STATE
       =SLOT1 =VALUE1
==>
   =GOAL>
       ACTION =ACTION
       ARG1 =ARG1
       ARG2 =ARG2
       VAL1 =VALUE1
       OPERATOR =ID
       EXP-SLOT =ESLOT
       EXP-NEG =ENEG
       EXP-VALUE =EVAL
       NEW-STATE =NEW-STATE
       FAIL =FAIL
   =STATE>
       =SLOT1 NIL
)
```



```
(P* MATCH-PRE1--STATE-SLOT-EQUAL-
   VALUE
   =GOAL>
       ISA GOAL
       TASK =TASK
   =RETRIEVAL>
       ISA OPERATOR
       TASK =TASK
       ID =ID
       ACTION =ACTION
       ARG1 =ARG1
       ARG2 =ARG2
       NEG1 "nil"
       SLOT1 =SLOT1
       VALUE1 =VALUE1
       EXP-SLOT =ESLOT
       EXP-NEG =ENEG
       EXP-VALUE =EVAL
       NEW-STATE =NEW-STATE
       FAIL =FAIL
   =STATE>
       ISA STATE
       =SLOT1 =VALUE1
==>
   =GOAL>
       ACTION =ACTION
       ARG1 =ARG1
       ARG2 =ARG2
       VAL1 =VALUE1
       OPERATOR =ID
       EXP-SLOT =ESLOT
       EXP-NEG =ENEG
       EXP-VALUE =EVAL
       NEW-STATE =NEW-STATE
       FAIL =FAIL
   =STATE>
       =SLOT1 NIL
)
```

A set of mismatch productions

```
(P* MATCH-NIL-IN-VAR-MISMATCH-1
   =GOAL>
       ISA GOAL
   =RETRIEVAL>
       ISA OPERATOR
       TASK =TASK
       ID =ID
       SLOT1 =SLOT
       NEG1 "nil"
       VALUE1 "nil"
   =VAR>
       ISA VARS
       =SLOT =ANY
 ==>
   !EVAL! (TRACE-OP =ID :FAIL T)
)
```

Actions:
pre-defined vs. self-defined

- ACT-R/Lisa comes with a set of predefined actions
- Sometimes some specific actions have to be added by the modeler

```
(p* implement-switch
   =goal>
        isa goal
        action switch
        arg1 =slot1
        arg2 =slot2
   =var>
       isa vars
       =slot1 =val1
       =slot2 =val2
==>
    =var>
       =slot1 =val2
       =slot2 =val1
    =goal>
        action done)
```

Finalizing productions

```
(p implement-done
   =goal>
        isa goal
        action done
        new-state "nil"
==>
   =goal>
        action nil
         arg1 nil
         arg2 nil
  !safe-eval! (setf (dm-finsts
   (get-module declarative))
   nil))
```

```
(p implement-done-newstate
   =goal>
    isa goal
    action done
    new-state =newstate
  !safe-eval! (not (equal =newstate
   "nil"))
==>
   =goal>
         state =newstate
         action nil
         arg1 nil
         arg2 nil
  !safe-eval! (setf (dm-finsts (get-
   module declarative)) nil))
```

The operator definition

(op *operator-name*
:goal *goal-name* (defaults to main goal)
:state *state* (defaults to start)
:pre ([-] *slot* [*value*])
:pre ([-] *slot* [*value*])
:action (*action* [*arg1*] [*arg2*])
:new-state *state*
:fail *state*
:post ([-] *slot* [*value*]))

Preconditions

- Operators can have up to two preconditions, which can match either the state or the var buffer
  - *slot* checks whether the slot has some value
  - (- *slot*) checks whether slot is empty
  - (*slot value)* checks whether slots equals value
  - (- *slot value*) checks whether slot and value are different

Action

- An action can have up to two arguments, but any values matched in the preconditions can also be used by the action
- Example:
-
```
(op store-aural
     :pre aural
     :action (store tone))
```
- This operator stores the value from the precondition (so whatever we heard) into variable tone

Post condition

- Is currently not used, except:
  - Spreads activation to retrieve next operator
- May be used in the future for planning strategies

Examples

Responding to perceptual and motor events	Subitizing
Instance retrieval	Paired Associates and Zbrodoff
Visual attention	Sperling
Multi-threading	Schumacher

Example model: Subitizing

```
(op first-item
   :pre visual-text
   :pre (- count)
   :action (set count "1"))
(op detect-next-item
   :pre count
   :pre visual-text
   :action (retrieve count-fact
   count))
(op report :pre visual-done :action
   (say count))
(op done-subitize
   :pre (vocal spoken)
   :action (repeat))
```

- Conditions for the operators are cued by perceptual events that are set in the state buffer:
  - visual-text: a text has been read
  - visual-done: everything on the display has been read
  - vocal spoken: a speak action has been completed

Example model: Paired associates

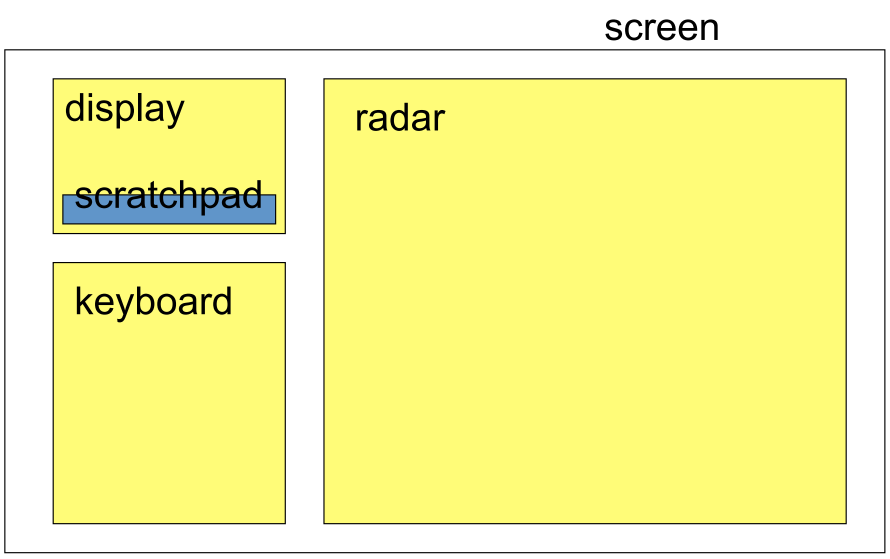
```
(op read-probe :action (read
   probe))
(op retrieve-key :pre probe
   :action (instance key)
   :fail feedback)
(op key-answer :action
   (press key)
   :new-state feedback)
(op read-feedback
   :state feedback
   :action (read key)
   :new-state done)
(op done-paired :state done
   :action (repeat))
```

- A repeat action renews the var buffer
- Old var chunks serve as instances
- The instance action tries to retrieve a var chunk matching all the slots in var that already have a value

Example: Zbrodoff

```
(op read-letter :action (read num1))
(op read-number :pre num1 :action (read num2))
(op try-instance :pre num2 :action (instance answer) :new-state decide :fail count)
(op copy-to-answer :state count :action (set answer num1))
(op increment-letter :state count :pre answer :action (retrieve count-fact answer)
    :new-state count2 :fail decide)
(op set-zero :state count2 :action (set count "1") )
(op increment-number :state count2 :pre (- count num2) :action (retrieve count-fact
    count) :new-state count :fail decide)
(op done-count :state count2 :pre (count num2)  :new-state decide)
(op answer-yes :state decide :pre (visual-text answer) :pre (- manual) :action (press
    "k"))
(op answer-no :state decide :pre (- visual-text answer) :pre (- manual) :action (press
    "d"))
(op done-zb :state decide :pre (manual press-key) :action (repeat))
```

Directing visual attention: regions



Visual regions in Sperling

```
(region bottom-row :top 180 :bottom 220)
(region middle-row :top 130 :bottom 180)
(region top-row :top 60 :bottom 130)

(add-fact r1 region-map 2000 top-row)
(add-fact r2 region-map 1000 middle-row)
(add-fact r3 region-map 500 bottom-row)

(op read-a-stimulus :pre1 visual-text :pre2 visual-loc :action (store-2 letter place))
(op store-letter :pre letter :action (memorize place letter))
(op tone-found :pre aural-text :action (retrieve region-map region) :new-state focus :fail done)
(op focus-on-region :state focus :action (focus-visual-attention region) :new-state start)
(op start-report :pre1 visual-done :pre2 region :action (retrieve-all region) :new-state done)
(op done-sperling :state done :action (repeat))
```

Multi-threading

- ACT-R/Lisa supports having multiple threads (Salvucci & Taatgen, submitted)
- Example: Schumacher task

```
(define-model-lisa schumacher1 (do-vm do-av) (word finger)

(fact av1 mapping 2000 "one")
(fact av2 mapping 1000 "two")
(fact av3 mapping 500 "three")

(op trigger-vm :goal do-vm :pre (- visual-text) :pre (- finger) :action (trigger
   visual-text))
(op see :goal do-vm :pre visual-text :action (extract finger))
(op press-finger :goal do-vm :pre (- manual) :action (punch finger))
(op done-vm :goal do-vm :pre (manual press-key) :action (clear finger))

(op trigger-av :goal do-av :pre (- aural-text) :pre (- word) :action (trigger
   aural-text))
(op hear :goal do-av :pre aural-text :action (Retrieve mapping word))
(op say-word :goal do-av  :pre (- vocal) :action (say word))
(op done-av :goal do-av :pre (vocal speak) :action (clear word))
```

Future directions

- Still very much work in progress
- More elaborate visual attention
- Time perception
- Handling of missing operators through subgoaling (Soar style!)
- Keep an eye on: http://www.ai.rug.nl/~niels/lisa.html

Integrating Architectures

Christian Lebiere
Carnegie Mellon University
cl@cmu.edu

Overview

- Motivation
- Levels
- Instances
- Issues
- Protocol
- Questions

Motivation

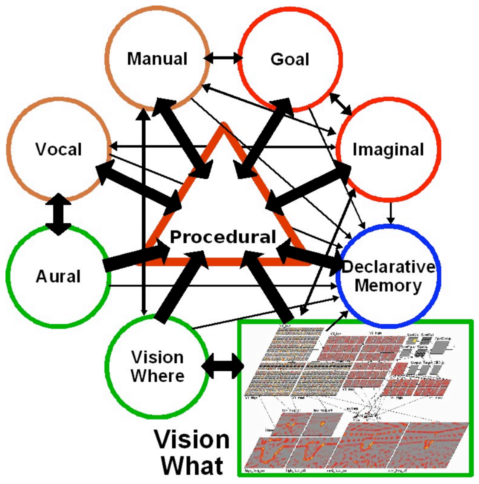
- Cognitive Architectures are general but…
  - They tend to be specialized to a class of problems
  - They tend to select a particular level of abstraction
- Linking architectures together is a solution to:
  - Broaden the class of applicable problems
  - Leverage multiple levels of mechanisms
    - Specific mechanisms are suitable for different problems
  - Provide abstraction to the lower-level components
    - Interact with the highest architectural level possible

Levels of Integration

- Loose Integration (focus of this tutorial)
  - Keep (pieces of) architectures relatively intact
  - Pro: easy to integrate and develop working prototype
  - Con: limited leverage, flexibility and grain-scale of integration
- Tight Integration
  - Merge the architectures constituent pieces and mechanisms
  - Pro: maximize power (no black box) and psychological plausibility
  - Con: integration is difficult and time-consuming
- The distinction is not always clear or constant:
  - ACT** integration of activation and production system?
  - ACT-R(/PM) integration of cognition and perception/motion?
  - General trend toward modular systems, i.e. loose integration

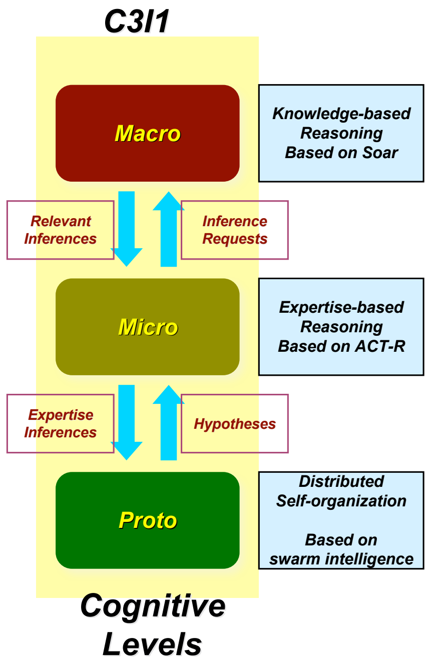
Instance 1: SAL

- SAL: Synthesis of ACT-R and Leabra
- Good fit w/ compatible modular approaches
- Several specific attempts at integration
- Display shows Leabra as ACT-R visual module



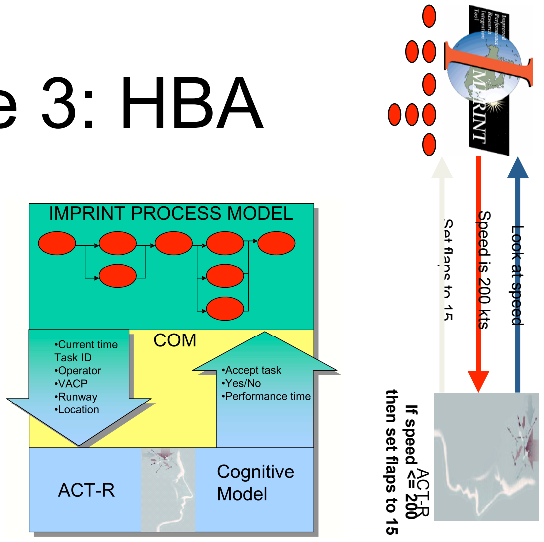
Instance 2: MCA

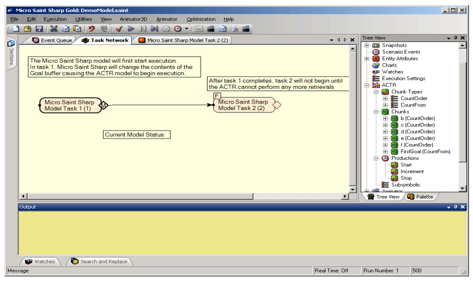
- Multi-level Cognitive Architecture (aka C3I1):
  - Swarm as Proto
  - ACT-R as Micro
  - Soar as Macro/Meta
- Basic principle of "Cognitive Pyramid" of trading data complexity for method tractability
- Loose toward tight integration


C3I1
Macro
Knowledge-based Reasoning Based on Soar
Relevant Inferences
Inference Requests
Micro
Expertise-based Reasoning Based on ACT-R
Expertise Inferences
Hypotheses
Proto
Distributed Self-organization Based on swarm intelligence
Cognitive Levels

Instance 3: HBA

- Human Behavior Architecture (HBA)
- Goal to make ACT-R more user-accessible
- Combine bottom-up cognitive architecture with top-down task decomposition
- 3 increasingly fine-grained implementations





The Micro Saint Sharp model will first start execution.
In task 1, Micro Saint Sharp will change the contents of the Goal buffer causing the ACTR model to begin execution.

After task 1 completes, task 2 will not begin until the ACTR cannot perform any more retrievals

Micro Saint Sharp Model Task 1 (1)

Micro Saint Sharp Model Task 2 (2)

Current Model Status:

Interaction Scale

- Coarse: procedure call
  - HBA: expand MicroSaint tasks into ACT-R goals
  - MCA: ACT-R calls Soar to solve an unrecognized goal
  - SAL: ACT-R calls Leabra to recognize one pattern
- Fine: constant interaction
  - HBA: productions react to each event
  - MCA: swarming constantly re-clusters knowledge base
  - SAL: ACT-R reacts to Leabra’s activation values

Time Synchronization

- Synchronous
  - Pass current time to reflect decay and similar factors
  - Duration of call passed back as argument
  - No expected interruption of client-server interaction
- Asynchronous
  - Each event is timestamped with current time
  - Each side executes in parallel
  - Must prevent one side from outpacing the other
  - Must be able to handle interrupts at any time
  - Difficult to engineer correctly

SAL Protocol

- Software framework
  - ACT-R module protocol as starting point. Too limited?
- Data formats
  - How is data exchanged across applications?
- Symbols and variables
  - Not essential but nice to handle abstract references
- Command formats
  - What are the valid commands and their arguments?
- Synchronization
  - How is the exchange structured and synchronized?

Software Framework

- Connection by text-based socket (& images?)
- Who establishes connection and on which port?
- Client-server relationship is symmetrical
  - Each side sends and handles commands
- Conversion encapsulated within each application
- Termination best handled explicitly
  - Closing socket not best way to wrap it up (retry?)
- Specific commands and semantics TBD

Communication Segmentation

- Line-based or packet-based?
- Each packet is one command
- Packet can span multiple lines
- Packet terminated by an empty line
- Line separator is CR/LF

Data Formats

- Can’t rely on internal representations
  - Not across languages and platforms
- Numbers
  - Integers, floats (precision, IEEE standards?)
- Strings
  - Double quotes, escape character \
- Arrays
  - Nested vectors (), arbitrary dimensions?
- Images
  - Text-based encoding, watch out for separators!

Symbols and Variables

- Symbols
  - No special character, non-case-sensitive
- Variables
  - Not same concept as traditional variables!
  - Designate constant but large values, e.g. images
  - Not essential but save real-time bandwidth
  - Can be implemented as tables of name (I.e. symbol) to value associations

Commands

- Symbol followed by argument-value pairs
- Pre-defined set of commands:
  - Init and stop
  - Send
  - Run and return
  - Error
- Each new application will define a new set of command or at least command arguments

Init

- First command sent by each side to initiate run
- Arguments include model, specific parameters:

```
<init> ::= init { <parameter> <value> }
<parameter> ::= symbol
<value> ::= [symbol | number | string | array ]
```

- Examples:

```
init name ACT-R version "r370"
init model vision default-size 1.0
```

Stop

- Terminate connection before closing
- Optional argument includes termination reason:

```
<stop> ::= stop {<reason>}
<reason> ::= a string
```

- Examples:

```
stop
stop "Model complete"
stop "Received irreconcilable error message"
```

Send

- Creates “variable” and binds it to value
- Arguments are symbol-value pair:

```
<send> ::= send <variable-name> <variable-value>
<variable-name> ::= symbol
<variable-value> ::= [symbol | number | string | array]
```

- Examples:

```
send image-file1 "foo/bar/baz/doodle.jpg"
send input-table (1.2 2.3 7.9 .004 -21.9 53.0)
send error_result1 "Oh No!"
```

Run

- Main command to run program and return results (or error)
- Arguments are program name and parameter-value pairs:

```
<run> ::= run <program> { <parameter> <value> }
<program> ::= symbol
<parameter> ::= symbol
<value> ::= [symbol | number | string | array | variable]
```

- Examples:

```
run reset_model
run encode_data data (1.0 2.3 4.5 7.8)
run retrieve_memory type visual-object color blue
    location (100 200)
```

Return

- Returns results from run command, or else signal error (next)
- Arguments are program name and optional result pairs:

```
<run> ::= return <program> { <parameter> <value> }
<program> ::= symbol
<parameter> ::= symbol
<value> ::= [symbol | number | string | array | variable]
```

- Examples:

```
return reset_model result success
return encode_data value "green"
return retrieve_memory object chunk51
       activation 1.75 time .334
```

Synchronization States

- Commands can only be sent in certain states:
  - Preinit: from start to exchange of init commands
  - Idle: init completed but no command pending
  - Busy: received command and computing result
  - Idle: sent command and waiting for result
- Different levels of enforcement and exclusion
- Different levels of acknowledgment
  - Whether to send ACK for every command?
- Similar to ACT-R module interaction

Error

- Signals problem with command short of termination
- Arguments are error description string and optional values:

```
<error> ::= error <detail> { <parameter> <value> }
<detail> ::= [string | variable]
<parameter> ::= symbol
<value> ::= [symbol | number | string | array | variable]
```

- Examples:

```
error "Problem encountered"
error ″parse command" incoming-packet "start model foo"
error "bad data" value 1.5 expecting 1.0
error error_result1
```

Sample Initialization

- A->L:init name "ACT-R 6" version 425
- L->A:init model vision
- A->L:run start_vision
- L->A:return start_vision version 1.0
- A->L:send image_file1 "foo/bar/baz/image1.jpg"
- A->L:send image_file2 "foo/bar/baz/image2.jpg"
- A->L:send raw_image1 "<base64 encoded .jpg file> ...

Sample Interaction

- A->L:run move_attention filename image_file1
- ux 100 uy 100 lx 150 ly 150 stop_when settled

- L->A:return move_attention item armor

- A->L:run move_attention image_data raw_image1
- stop_when cycles cycles 75 ux 25 uy 75 lx 100 ly 140

- L->A:return move_attention item unidentified
- output_layer (0.05 -1.24 -0.95 ...)

- A->L:stop "Model has finished"

Advanced Issues

- Learning representation
  - How to deal with the fact that the representation mapping cannot be specified in advance?
  - Similar problem in ACT-R between modules
- Similarities and differences between integrating architectures vs. environments
  - Most problems (data exchange, time synchronization, etc) are largely similar
  - Semantics of commands substantially different from run/results to perception/action

Other Integration Targets

- Large knowledge bases, e.g. Cyc
  - Ontological commitments
  - Memory access model
- Language sources, e.g. WordNet
  - Subsymbolic instantiation
- External AI tools, e.g. parser, planner
  - Time course of processing?