Extending ACT-R

Dan Bothell

Updated slides originally

Presented at ICCM in 2007

Before we start

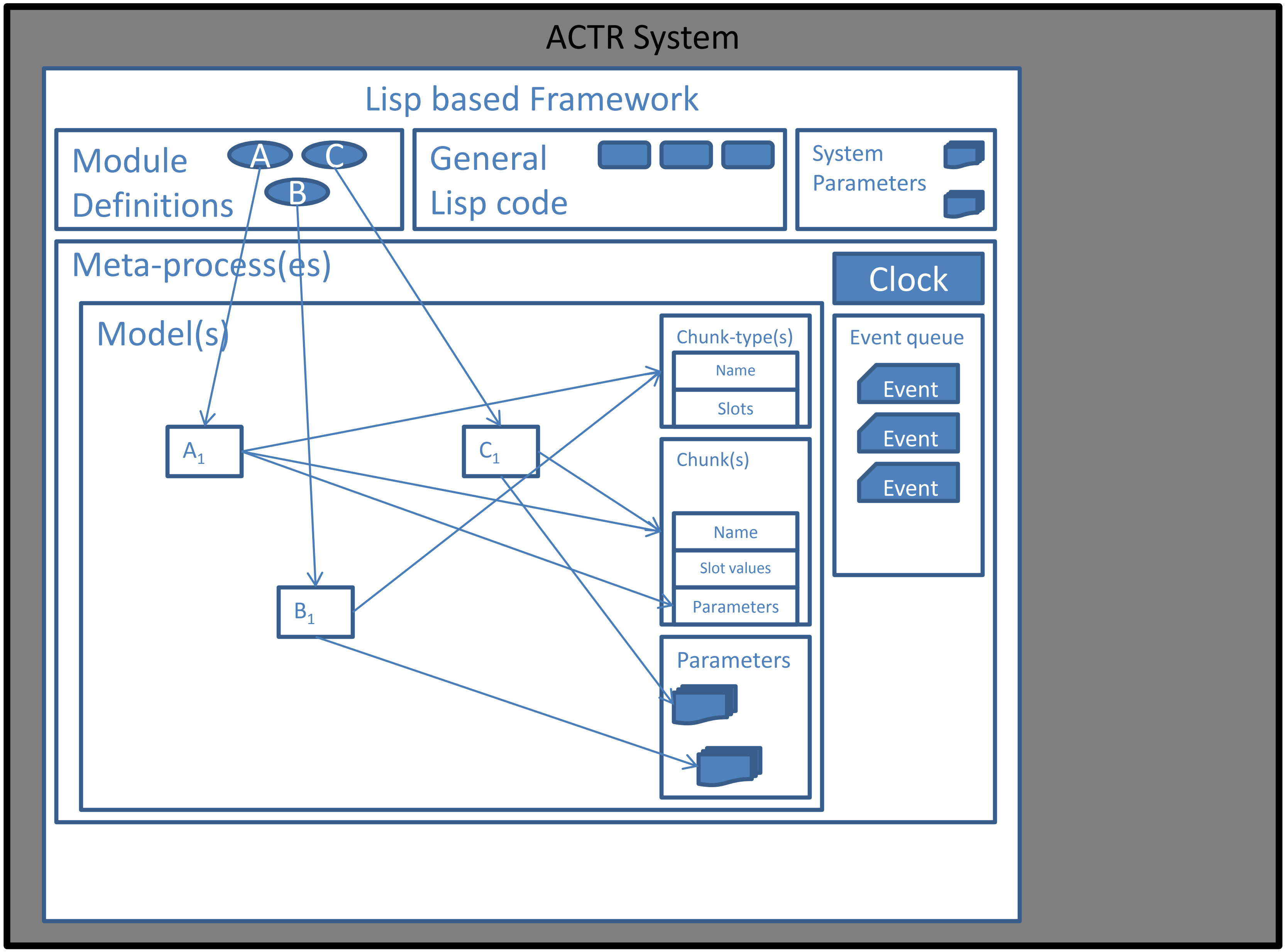
- Talking about the ACT-R software system
  - Not the theory
  - The software was designed for the theory
  - Not generally constrained by the theory or cognitive plausibility

- Assume that you can program in Lisp
  - At least able to read Lisp code
- Assume you can model in ACT-R
  - Familiar with the general operation of the system
  - Know the basic commands

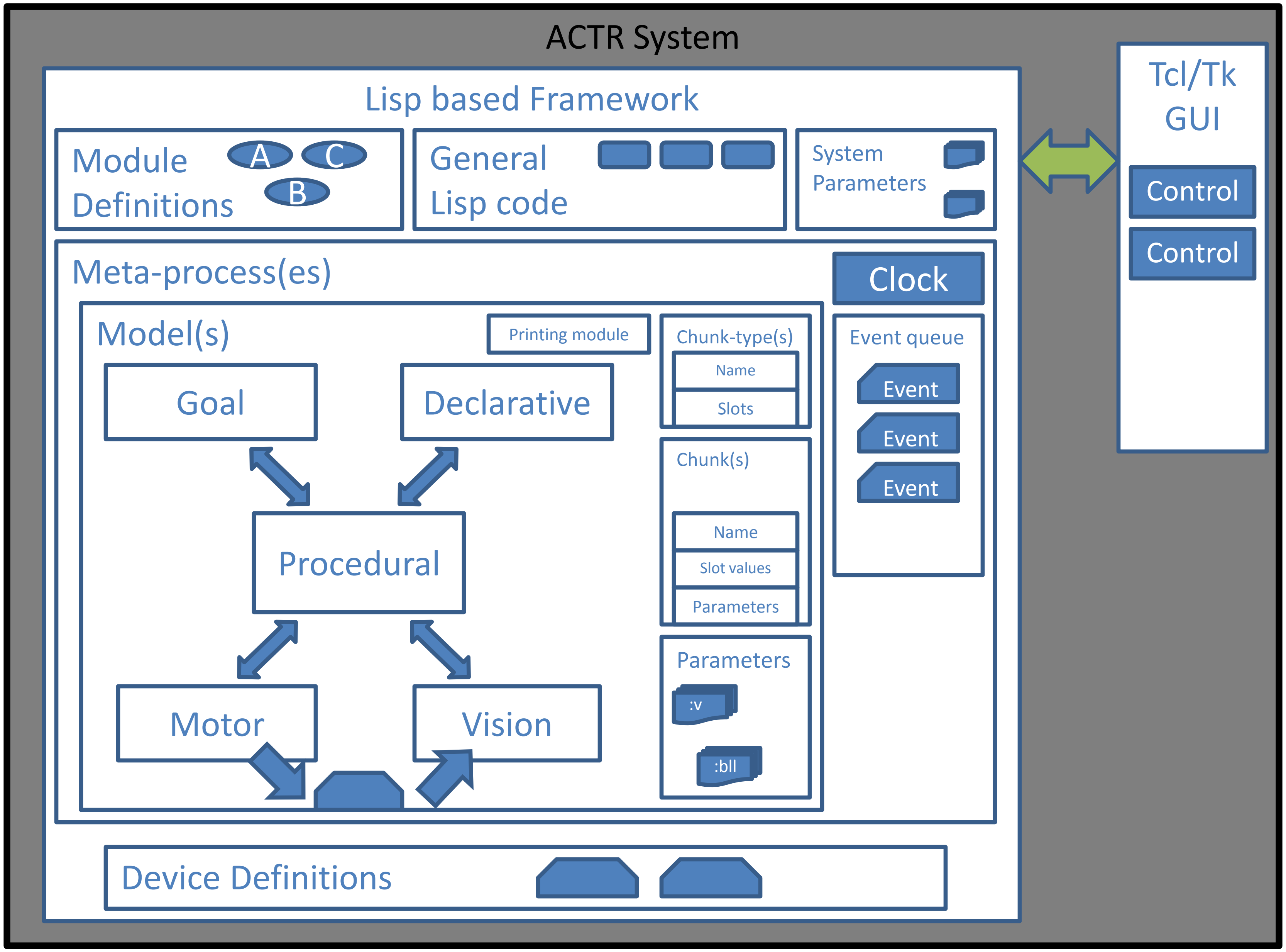
Outline

- Software overview
- Extensions
- Creating Devices
- Defining modules

ACT-R software (general)



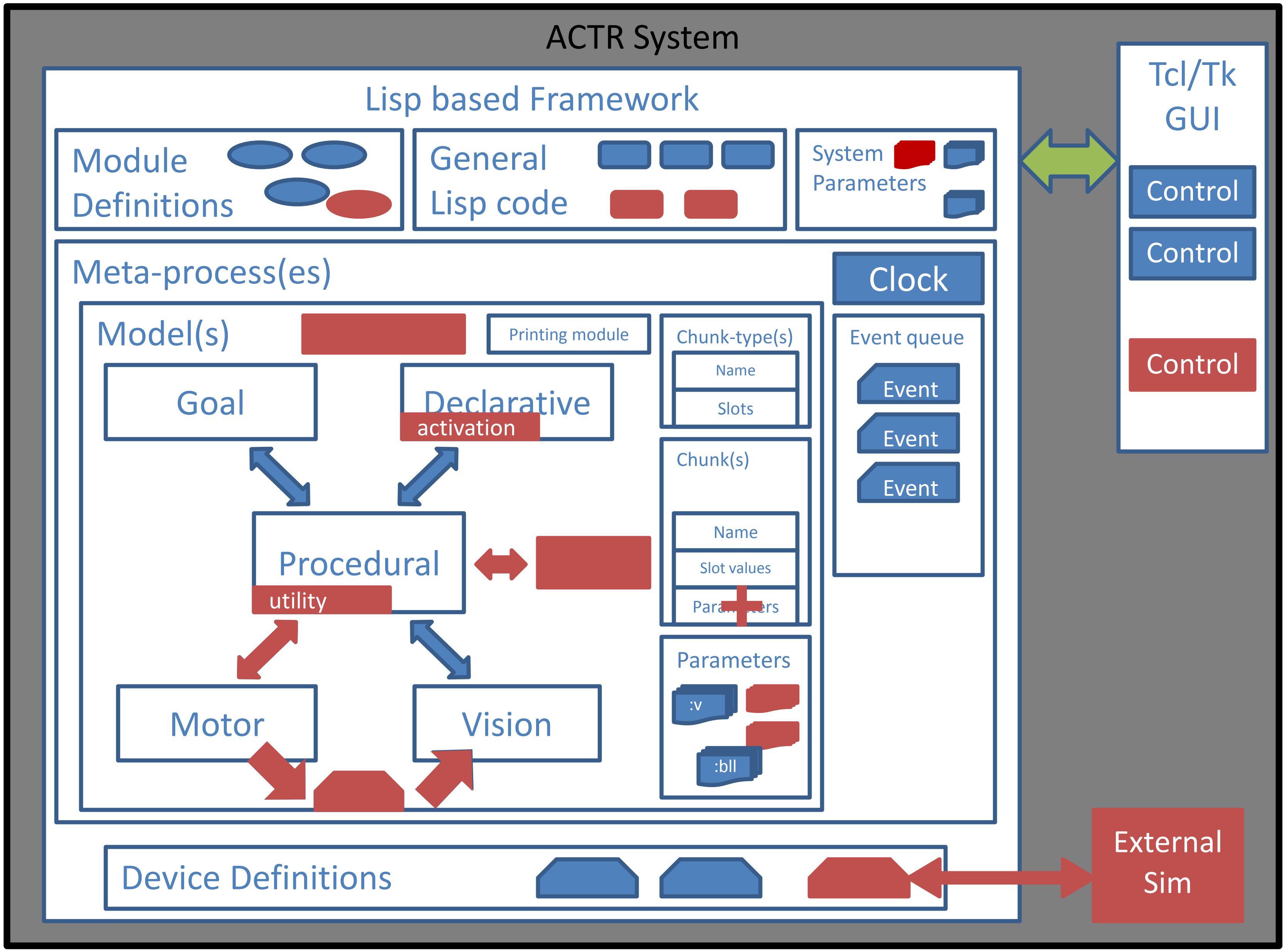
ACT-R software (specific)



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 the Reference manual

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

- Build-vis-locs-for
- Cursor-to-vis-loc
- Vis-loc-to-obj
- Vis-loc-coordinate-slots (optional)

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 have at least an x and y coordinate (default visual-location type slots screen-x and screen-y)
  - Values don't matter
  - May have any other slots

- 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 has slots for the x and y position (like build-vis-locs-for)

- 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

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

Vis-loc-coordinate-slots

- Optional method called when a device is installed
  - If not provided the default screen-x, screen-y, and distance slots will be used
- Passed one value
  - The current device
- Must return a list of three symbols

- Those symbols name the slots which the visual-location chunks for this device will use to hold the x, y, and z coordinates for items repsectively

Creating a new device

- Will use a list as the device itself
  - 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 and the default coordinate slots will be used
  - Runnable example available in the sources:

    examples/creating-devices/simple-new-device.lisp

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 (polygon t))
  (chunk-type (square (:include polygon)) (sides 4) (square t))

  (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
  - 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 in the examples/creating-devices directory
  - 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 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
- Complete code for this module available in the examples/creating-modules directory
  - demo-module.lisp and demo-model.lisp

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

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

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

```
>(mp-print-versions)
ACT-R Version Information:
…
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 reference manual

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 primary 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
  - every time a model is reset
  - after the initial creation (or other times if additional reset functions are specified – see manual for details)
- 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 will create a chunk-type for our output request

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

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*) or buffer test (LHS =*buffer*)
  - 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
- The chunk-spec does not include the optional isa component of a request

- 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
```

```
> (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)
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 ((output-slot? (chunk-spec-slot-spec spec 'demo-output))
        (v1 (chunk-spec-slot-spec spec 'value)))
     (if output-slot?
         (if v1
             (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 "demo-output slot missing 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*)
(SCREEN-X 10 SCREEN-Y 20)
> (define-chunks-fct (list (chunk-spec-to-chunk-def *foo*)))
(CHUNK2)
>  (pprint-chunks chunk2)
CHUNK2
  SCREEN-X  10
  SCREEN-Y  20
```

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 reference 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

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)
           (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 reference manual 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
      ?create>
      state free
      buffer empty
      ==>
      +create>
      isa visual-location
      screen-x 10
      screen-y 20)
```

```
(p p2
     =create>
     isa visual-location
     ==>
     +output>
     isa demo-output
     value =create
     ))
```

The trace (load time)

```
; Loading C:\actr6.1\examples\creating-modules\demo-model.lisp
 0[5]: (DELETE-DEMO-MODULE #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL))
 0[5]: returned NIL
 0[5]: (CREATE-DEMO-MODULE TEST-DEMO-MODULE)
 0[5]: returned #S(DEMO-MODULE :DELAY NIL :ESC NIL :BUSY NIL)
 0[5]: (RESET-DEMO-MODULE #S(DEMO-MODULE :DELAY NIL :ESC NIL :BUSY NIL))
 0[5]: returned DEMO-OUTPUT
 0[5]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY NIL :ESC NIL :BUSY NIL)
                            (:CREATE-DELAY . 0.1))
 0[5]: returned 0.1
 0[5]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :ESC NIL :BUSY NIL)
                            (:ESC))
 0[5]: returned NIL
 0[5]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :ESC NIL :BUSY NIL)
                            (:ESC . T))
 0[5]: returned T
 0[5]: (DEMO-MODULE-PARAMS #S(DEMO-MODULE :DELAY 0.1 :ESC T :BUSY NIL)
                            (:CREATE-DELAY . 0.15))
 0[5]: returned 0.15
```

The trace (run time)

```
CG-USER(74): (run .25)
     0.000   PROCEDURAL             CONFLICT-RESOLUTION
 0[5]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL) CREATE STATE FREE)
 0[5]: 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[5]: (DEMO-MODULE-REQUESTS #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL) CREATE
                             #S(ACT-R-CHUNK-SPEC...))
 0[5]: returned #S(ACT-R-EVENT ...)
     0.050   PROCEDURAL             CLEAR-BUFFER CREATE
     0.050   PROCEDURAL             CONFLICT-RESOLUTION
 0[5]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY T) CREATE STATE FREE)
 0[5]: returned NIL
     0.200   DEMO                   SET-BUFFER-CHUNK CREATE CHUNK0
     0.200   DEMO                   FREE-DEMO-MODULE #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY T)
     0.200   PROCEDURAL             CONFLICT-RESOLUTION
 0[5]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL) CREATE STATE FREE)
 0[5]: returned T
```

```
     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[5]: (DEMO-MODULE-REQUESTS #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL) OUTPUT
                             #S(ACT-R-CHUNK-SPEC ...))
Value: CHUNK0-0
 0[5]: returned NIL
     0.250   PROCEDURAL             CLEAR-BUFFER CREATE
     0.250   PROCEDURAL             CLEAR-BUFFER OUTPUT
     0.250   PROCEDURAL             CONFLICT-RESOLUTION
 0[5]: (DEMO-MODULE-QUERIES #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL) CREATE STATE FREE)
 0[5]: returned T
     0.250   PROCEDURAL             PRODUCTION-SELECTED P1
     0.250   PROCEDURAL             QUERY-BUFFER-ACTION CREATE
     0.250   ------                 Stopped because time limit reached
0.25
23
NIL

> (clear-all)
 0[5]: (DELETE-DEMO-MODULE #S(DEMO-MODULE :DELAY 0.15 :ESC T :BUSY NIL))
 0[5]: returned NIL
NIL
```