ACT-UP
Documentation

David Reitter

June 25, 2009

1 Overview

ACT-UP is a cognitive modeling library that allows modelers to specify their model's functionality in Common Lisp. Whenever a cognitive explanation in a particular part of the model is sought, the modeler uses the library to provide characteristics of

- explicit, declarative learning and cue- and similarity-based retrieval, and
- procedural skill acquisition [not yet available]

following the ACT-R 6 theory.

As in the ACT-R 6 implementation, modelers are free to adhere more or less to the theoretical limitations. However, ACT-UP's design encourages modelers to underspecify portions of the model's functionality that do not contribute to the model's explanations and predictions of human performance.

2 How do I...

... load the library?

```
(require "act-up" "act-up.lisp")
(use-package :act-up)
```

... define a chunk type? Chunk types are lisp structure types that inherit from the type 'chunk'. For example, the following structure defines a chunk type of name 'strategy' with four slots. One of these slots is assigned a default value ('strategy').

```
(define-chunk-type strategy
  (type 'a-strategy)
  name
  dampen
  success
)
```

Note that the 'type' member is not required by ACT-UP.

To define an inherited type, use this construction:

```
(define-chunk-type (lazy-strategy :include strategy)
...
```

...define a new model? The model is defined automatically when 'act-up.lisp' is loaded. To reset the model, use the 'reset-model' function. To create a new model (multiple models may be used in parallel), use '(make-model)'. Use the function 'set-current-actUP-model' to define the current model.

The ACT-UP meta-process keeps track of model time that is common to all models. You may define several meta-processes and use/reuse them as you like with the function 'make-meta-process'. You can bind *current-actUP-meta-process* to a meta-process to switch. Use 'reset-mp' to discard and reset the current meta-process.

...commit a chunk to memory or reinforce it? To specify the "presentation" of a specific chunk, use the function 'learn'. The chunk reference may be supplied in a normal variable (equivalent to ACT-R's buffer), or the chunk may be produced right there and then using the 'make-type' syntax , as in the following example:

```
(learn-chunk  (make-strategy :name 'guess :success 0.2))
```

This will create a new strategy chunk, setting two of its parameters, and commit it to memory. Note that even newly created chunks will be merged with existing chunks if they contain the same elements.

... retrieve an item from declarative memory? Simply use the high-level functions 'retrieve-chunk', or 'blend-retrieve-chunk' (for blending). The following example retrieves the most active chunk that has the name 'guess' . The chunk contained in the variable 'valve-open-chunk' spreads activation. No partial matching is used:

```
(retrieve-chunk '(:name guess))
    (list valve-open-chunk)
    nil)
```

Several low-level functions are provided as well. 'filter-chunks' produces a list of all chunks that match a given set of criteria. In the example below, we are looking for a chunk with the 'name' attribute 'guess'.

The 'best-chunk' function does the actual (time-consuming and noisy) retrieval: it selects the best chunk out of the (filtered) list of chunks, given additional retrieval cues that spread activation and, if so desired, a set of filter specifications for partial matching. In this example, we use an existing chunk stored in the 'valve-open-chunk' variable as a single retrieval cue, and no partial matching:

```
(best-chunk (filter-chunks
                 (model-chunks (current-actUP-model))
 '(:name guess))
    (list valve-open-chunk)
    nil)
```

... retrieve a blended chunk? Use the high-level function 'blend-retrieve-chunk' .

When combining low-level functions, use the function 'blend' instead of 'retrieve-chunk'. In addition to the cues and partial-matching specification known from 'retrieve-chunk', it also expects a chunk type (such as 'strategy'), which determines the kind of chunk created as a result of blending.

... define chunk similarities? Use the 'add-sji-fct' and reset-sji-fct' functions.

... define a procedural rule ("production")? ACT-UP does not use if-then production rules as known from ACT-R. Instead, it allows you define Lisp functions. However, rather than through Lisp's 'defun' macro, you define rules using 'defrule':

```
(defrule subtract-digit (minuend subtrahend)
   "Perform subtraction of a single digit"
   (- minuend subtrahend))
```

... select a procedural rule using subsymbolic utility learning? Define competing rules as above and give each a `:group` attribute in order to group them into a competition set:

```
(defrule  force-over ()
  :group choose-strategy
  ...)
(defrule force-under ()
  :group choose-strategy
  ...)
```

Then, invoke one of the rules (as chosen by utility) as such:
`(choose-strategy)`

Arguments may be used as well (but ensure that all rules accept the same arguments).

Utilities are learned using the function 'assign-reward':
`(assign-reward 1.5)`

This example distributes a reward of 1.5 across the recently invoked rules. Rules do not have to have a `:group` attribute and they do not have to have been invoked via the group name in order to receive a reward; however, they

have to have been defined using the 'defrule' macro (rather than just being Lisp functions).

Configure utility learning via the parameters '*au-rpps*', '*au-rfr*', '*alpha*', and '*iu*'.