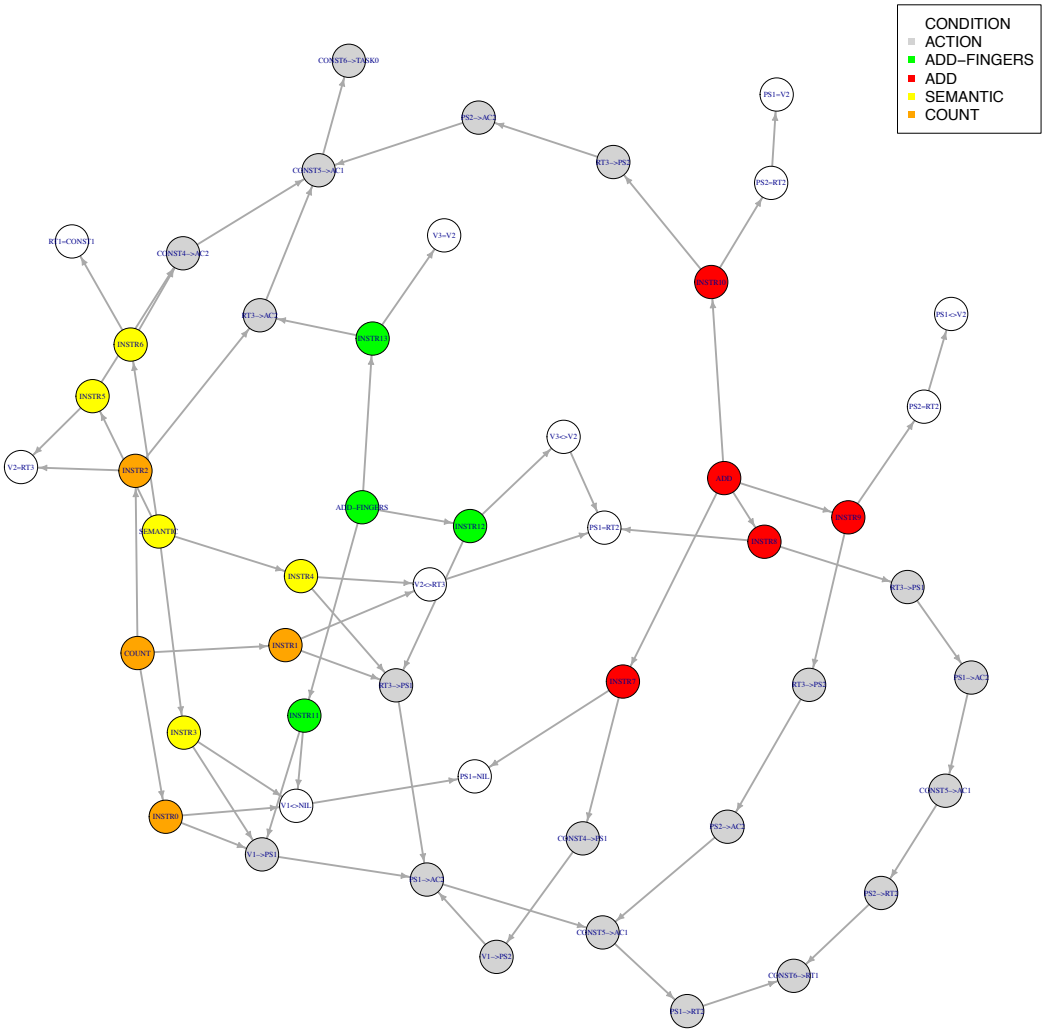
ACTRANSFER TUTORIAL



Funded by ERC StG
283597 MULTITASK

Niels Taatgen
July 2012 (Updated January 2014)

Introduction

This document is a tutorial to get started with Actransfer, an extension to ACT-R to model transfer phenomena. An example using addition and semantic reasoning will be discussed, followed by an exercise in which you have to add a model that does addition.

If this document is not already part of the tutorial package, retrieve a copy of actransfer and the tutorial files from the following webpage:

http://www.ai.rug.nl/~niels/actransfer/tutorial.html

In addition to the files specific to Actransfer, there is a version of ACT-R with which the system is guaranteed to work (currently July 2012).

To start using Actransfer, simply compile and load the file after loading ACT-R itself. You can then load an Actransfer model, just as you would load a regular ACT-R model.

Actransfer has been tested on Lispworks and Clozure CL, and has been reported to work with Allegro CL.

The structure of an Actransfer model

Basically the only thing an Actransfer model provides is a specification of task instructions or operators in ACT-R's declarative memory. Given that these operators are hard to interpret in the regular ACT-R syntax, a small specification language is provided that makes things more readable. In addition to the model code itself, a model file typically contains, as is common in all ACT-R models, Lisp code to simulate the experiment and to collect data.

The example we will use consists of two models from the standard unit 1 of the ACT-R tutorial, count and semantic, but now translated into Actransfer. The operators in Actransfer correspond one-on-one with productions in the unit 1 models.

Let us look at the file CountSemantic.lisp. After a number of Lisp functions, the model starts with:

```
(define-model-transfer
(add-dm
;;; count-facts
 (count0 isa fact slot1 count-fact slot2 zero slot3 one)
 ...
 (count5 isa fact slot1 count-fact slot2 five slot3 six))
```

This is just like in a regular ACT-R model: we start the model definition with some declarative facts. Note though that all facts are of type "fact", and use slot names slot1..slot*n* (*n*<5, presently). After DM declarations, the "real" model starts:

```
(add-task count :input (Vstart Vend) :working-memory (WMcount) :declarative ((RTcount-
   fact RTfirst RTsecond))

:pm-function do-action

:init init-count

:reward 10.0

:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -0.5 :alpha 0.2))

  (operator :condition (Vstart<>nil WMcount=nil) :action (Vstart->WMcount
   (say WMcount)->AC  (count-fact WMcount)->RT) :description "Initialize Count")

  (operator :condition (Vend<>RTsecond  WMcount=RTfirst) :action (RTsecond->WMcount
   (say WMcount)->AC (count-fact WMcount)->RT) :description "Counting Step")

  (operator :condition (Vend=RTsecond) :action ((answer RTsecond)->AC
   finish->Gtask) :description "Finalize count"))
```

The function "add-task" starts the declaration of a model for a new task, in this case "count". After giving the task a name, it is followed by a number of declarations that each start with a colon (:).

The first three of these declarations give names to the slots in the buffers. These declarations are comparable to chunk-types, except that they don't actually end up in the declarative memory representations themselves, but are replaced by generic slots slot1..slot*n*. The input buffer (which is used for all perceptual input) has two slots: Vstart and Vend. We will use these to set the start and end numbers for counting. In the counting task, the input is just set to particular numbers, but in a more complex model these values can change depending on the model's actions (or because things just happen in the world). Even though you can use any identifier that you like to refer to the slots, we will use the convention of starting with uppercase letter(s) that indicates the buffer.

Working memory refers to the problem-state or imaginal buffer :working-memory (WMcount) names a slot in the problem-state buffer. There can be up to four slots in the problem-state buffer. "Declarative" refers to facts retrieved from memory. It is a list, because there can be multiple different types. Again, types are only for our convenience, because they are translated to generic slot names in the actual model. In the count example, there are only count-facts.

The next two declarations, :pm-function and :init, specify which Lisp functions interface with the experimental code. I will explain those later.

The reward parameter (:reward) specifies the reward the model gets when successful. Finally, the :parameters declarations allows you to set any ACT-R parameters, or execute any other Lisp code during the loading of the model (whatever is in parentheses is just evaluated).

The essence of the model, though, is in the operators. They all start with "(operator", and are followed by a list of conditions and a list of actions. Optionally you can add a description that will be printed in the trace (strongly recommended!). On the condition side, there are four types of comparisons that can be made: `BufferSlot1=BufferSlot2`, `BufferSlot1<>BufferSlot2`, `BufferSlot1=nil`, or `BufferSlot1<>nil`. You can use any of the slot names you have defined in these tests, and, in addition, four predefined slots: Gtask, Gcontrol, Gparent and GWM. Gtask refers to a slot in the goal that represents the current task (count in the example). If you change that slot, you'll change the task you are doing. Gcontrol is also a slot in the goal that can be used to represent a control state. We will not discuss Gparent and GWM here, because they are used for working memory control, which we will not touch upon in this tutorial.

Instead of a BufferSlot you can also put a constant value in a comparison, which will be in lowercase by convention. For example, we can check whether the count in working memory is zero by checking `WMcount=zero`. In the model, constants will be put in one of the buffers, so `WMcount=zero` will be translated to a comparison between an imaginal buffer slot and a slot in the goal that holds the constant zero. A condition involving any BufferSlot will only be satisfied if the value of that Slot is non-nil, with the exception of `Bufferslot1=nil`, of course. So `WMcount<>Vend` will not match if `WMcount` is still nil.

There is only one type of action, and that is one in which you copy the contents of one slot to another. The format for this is `BufferSlot1->Bufferslot2`. For some buffers it is convenient to specify multiple steps at the same time. For example, instead of `count-fact->RTcount-fact WMcount->RTfirst`, you can specify `(count-fact WMcount)->RT`. Again, this is just syntax and will lead to the same declarative representation. On the action side there is an addition buffer "AC", which is used to represent external actions (manual, vocal, etc). You will typically copy all values to the AC in one step (like with RT), so the individual slots have no specific names.

To got a sense of how an operator specification translates into declarative memory let us look at an example: the first operator in the counting model.

```
(operator :condition (Vstart<>nil WMcount=nil) :action (Vstart->WMcount
   (say WMcount)->AC (count-fact WMcount)->RT) :description "Initialize Count")
```

This operator translates into declarative chunks in the table below (note that you don't need to understand this translation for the tutorial). What you can see is that all the slotnames

have been replace by generic names (e.g., V1 instead of Vstart), and that all constants have been put in INSTR0 (i.e., SAY and COUNT-FACT). These are referred to with CONST5 and CONST6, respectively. The table also already shows one chunk (INSTR3) of the semantic task that we will discuss later.

```
(operator :condition (Vstart<>nil WMcount=nil)
   :action (Vstart->WMcount  (say WMcount)->AC (count-fact WMcount)->RT)
   :description "Initialize Count")
```

```
(INSTR0
   ISA INSTR
   TASK  COUNT
   CONDITION  CD1
   ACTION  AC4
   SLOT1  NULL
   SLOT2  NULL
   SLOT3  NULL
   SLOT4  NULL
   SLOT5  SAY
   SLOT6  COUNT-FACT)
```

```
CD1
  ISA CONDITION
   NAME  CD1
   C  V1<>NIL
   CNEXT  CD0
   SLOT1  NIL
```

```
CD0
  ISA CONDITION
   NAME  CD0
   C  PS1=NIL
   CNEXT  CSTOP
   SLOT1  NIL
```

```
AC4
  ISA ACTION
   NAME  AC4
   A  V1->PS1
   ANEXT  AC3
   SLOT1  NIL
```

```
AC3
  ISA ACTION
   NAME  AC3
   A  PS1->AC2
   ANEXT  AC2
   SLOT1  NIL
```

```
INSTR3
  ISA INSTR
   TASK  SEMANTIC
   CONDITION  CD1
   ACTION  AC4
   SLOT1  NULL
   SLOT2  NULL
   SLOT3  NULL
   SLOT4  NULL
   SLOT5  SAY
   SLOT6  PROPERTY
```

```
AC2
  ISA ACTION
   NAME  AC2
   A  CONST5->AC1
   ANEXT  AC1
   SLOT1  NIL
```

```
AC1
  ISA ACTION
   NAME  AC1
   A  PS1->RT2
   ANEXT  AC0
   SLOT1  NIL
```

```
AC0
  ISA ACTION
   NAME  AC0
   A  CONST6->RT1
   ANEXT  ASTOP
   SLOT1  NIL
```

The example operator in the table initializes the count, and is applicable when the counter in working memory has not been set yet, but when there is a perceptual input the specifies the starting number. Therefore the conditions are `Vstart<>nil`: there is something in the input, and `WMcount=nil`: the counter is still nil. If both conditions apply, there are three actions (or actually five, because the last two consists of two actions): `Vstart->WMcount`, move the perceptual input into the counter in working memory, `(say WMcount)->AC`, move the constant "say" and contents of the counter to the Action buffer (so these are actually two actions: copying "say" and copying the counter). The action buffer forwards whatever you put into it to your self-defined pm-function, which than is assumed to carry

out the action. We will discuss this function in more detail later. The final condition is a retrieval for the number after the current count: `(count-fact WMcount)->RT`. This specification also translates into two actions, in which the count-fact constant is put into the first slot of the retrieval request, and the value in WMcount in the second retrieval slot.

Note that, unlike in ACT-R, conditions are tested serially, and actions are carried out in the order listed.

The second and third operator do the rest of the counting: the second iterates as long as the final number has not been reached, and the third operator terminates the count. Here we see and example of the special Gtask slot: by setting the task to finish (`finish->Gtask`), we end the task.

Running a model

To run a model, load in the model file. You can run the model "manually" with the following three commands:

```
(set-task 'count)
(init-task)
(run 100)
```

By default, standard ACT-R tracing is off, because it will, even for this small model, produce a long trace. You can switch on regular ACT-R tracing `(sgp :v t)` to see all the details though (or looking at the buffer trace in the Environment after switching on :save-buffer-trace) Instead, it is better to switch on Actransfer's tracing `(setf *verbose* t)`, and ACT-R's tracing off `(sgp :v nil)`. The function

```
(do-count 1)
```

will do all that, and produce the following trace:

```
***     0.34: INSTR0: Initialize Count
***     1.57: ACTION: SAY TWO
***     1.69: Retrieving fact COUNT2: COUNT-FACT TWO THREE NIL
***     2.02: INSTR0: Initialize Count
***     2.30: Mismatch on condition CD0-1
***     2.41: INSTR1: Counting Step
***     3.79: ACTION: SAY THREE
***     3.82: Retrieving fact COUNT3: COUNT-FACT THREE FOUR NIL
***     4.19: INSTR2: Finalize count
***     5.06: ACTION: ANSWER FOUR
```

You can get some more detail in this trace by setting *verbose* to full (setf *verbose* 'full). Run (test-count) to see the result of that:

```
***     0.35: INSTR0: Initialize Count
            0.49: Testing condition CD1: V1<>NIL
            0.61: Testing condition CD0: PS1=NIL
            0.73: All conditions matched
            0.89: Carrying out action AC4: V1->PS1
            1.02: Carrying out action AC3: PS1->AC2
            1.13: Carrying out action AC2: CONST5->AC1
            1.24: Carrying out action AC1: PS1->RT2
            1.36: Carrying out action AC0: CONST6->RT1
            1.49: All actions done
***     1.59: ACTION: SAY TWO
***     1.70: Retrieving fact COUNT2: COUNT-FACT TWO THREE NIL
***     1.98: INSTR1: Counting Step
            2.11: Testing condition CD3: V2<>RT3
            2.23: Testing condition CD2: PS1=RT2
            2.40: All conditions matched
            2.59: Carrying out action AC5: RT3->PS1
            2.72: Carrying out action AC3: PS1->AC2
            2.86: Carrying out action AC2: CONST5->AC1
            2.99: Carrying out action AC1: PS1->RT2
            3.11: Carrying out action AC0: CONST6->RT1
            3.23: All actions done
***     3.38: ACTION: SAY THREE
***     3.42: Retrieving fact COUNT3: COUNT-FACT THREE FOUR NIL
***     3.79: INSTR2: Finalize count
            3.91: Testing condition CD4: V2=RT3
            4.08: All conditions matched
            4.29: Carrying out action AC8: RT3->AC2
            4.43: Carrying out action AC7: CONST5->AC1
            4.55: Carrying out action AC6: CONST6->TASK0
            4.68: All actions done
***     4.78: ACTION: ANSWER FOUR
```

In this trace all the individual conditions and actions are listed.

If we run the model multiple times, for example (do-count 10), we can see that the model gradually speeds up due to production compilation:

```
 4.835
 5.003
 4.804
 4.837
```

```
4.842
4.438
4.323
4.843
4.816
4.116
```

If you run the model multiple times, steps will start dropping out of the trace, because items in the trace are triggered by a retrieval. Run the model 40 times (do-count 40), and then run (test-count) again. You will see that many of the steps have dropped out, and the model is quite a bit faster.

Modeling Transfer

The goal of Actransfer is, of course, to model transfer. We therefore need to specify at least one additional model. The example is the Semantic model from unit 1. The goal of the semantic model is to judge relationships between animals and animal categories, and answer question like "Is a canary an animal"? The facts the model uses are:

```
(p1 isa fact slot1 property slot2 canary slot3 bird)
(p2 isa fact slot1 property slot2 shark slot3 fish)
(p3 isa fact slot1 property slot2 bird slot3 animal)
(p4 isa fact slot1 property slot2 fish slot3 animal))
```

Answering the question involves two steps: first to retrieve that a canary is a bird, and then that a bird is an animal. Even though this model is semantically different from count, it shares the same type of iteration. The model is therefore similar:

```
(add-task semantic :input (Vanimal Vcategory) :working-memory (WMcurrent) :declarative
   ((RTprop RTitem RTmember-of))

:pm-function do-action

:init init-semantic

:reward 10.0

:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -0.5  :alpha 0.2))

  (operator :condition (Vanimal<>nil WMcurrent = nil) :action (Vanimal->WMcurrent
   (say WMcurrent)->AC  (property WMcurrent)->RT)
   :description "Retrieve first category")

  (operator :condition (Vcategory<>RTmember-of  WMcurrent = RTitem)
   :action (RTmember-of->WMcurrent  (say WMcurrent)->AC
   (property WMcurrent)->RT) :description "Chain up")

  (operator :condition (Vcategory = RTmember-of)
   :action ((answer yes)->AC finish->Gtask) :description "Match found")

  (operator :condition (RTprop = error)
   :action ((answer no)->AC finish->Gtask) :description "No Match")))
```

The first two operators in this model are the same as in the counting model, with the exception that the slot labels are different. But once the operators are translated into declarative memory, the conditions and actions are identical (we can see this in the table on page 5). The last two operators are different. If a match between the target category and the retrieved category is found, the model should answer "yes", which is slightly different from finalizing the count. Also, if the proposition does not hold, the model will hit a retrieval error at some point. On a retrieval error, the first slot of the retrieval buffer will be set to error (which is, in the example, matched by (RTprop=error).

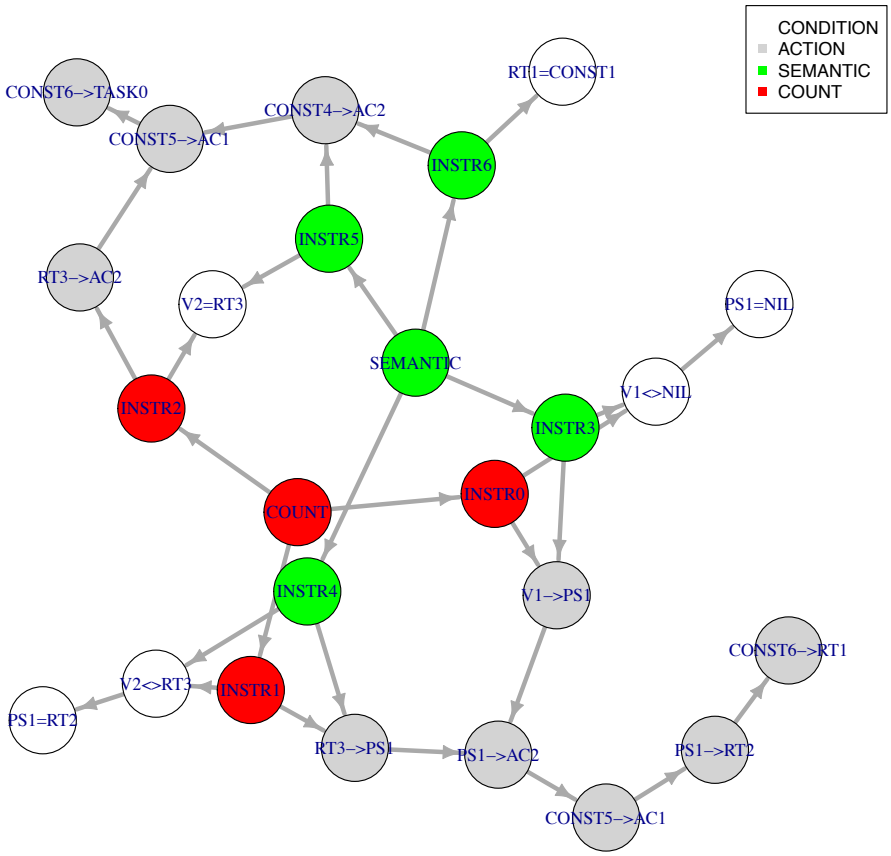
Here is an example of a run of the model (through (do-semantic 1)):

```
***     0.63: INSTR3: Retrieve first category
***     1.98: ACTION: SAY CANARY
***     2.04: Retrieving fact P1: PROPERTY CANARY BIRD NIL
***     2.39: INSTR4: Chain up
***     3.72: ACTION: SAY BIRD
***     3.84: Retrieving fact P3: PROPERTY BIRD ANIMAL NIL
***     4.12: INSTR5: Match found
***     5.04: ACTION: ANSWER YES
```

How do we assess transfer between these two models? A first option is to look at the overlap of chunks in declarative memory between the two models. The figure on the next page gives an impression of this overlap, which is considerable.

To get a real sense of the amount of transfer, we have to run the model. We first run the Count model a number of times, and then the Semantic model. To get some insight into transfer, first run the count model 50 times (do-count 50), and then try out the semantic model (test-semantic):

```
***   222.87: INSTR3: Retrieve first category
          223.01: Testing condition CD0: PS1=NIL
          223.12: All conditions matched
          223.28: Carrying out action AC3: PS1->AC2
          223.40: Carrying out action AC0: CONST6->RT1
***   223.45: ACTION: SAY CANARY
***   223.58: Retrieving fact P1: PROPERTY CANARY BIRD NIL
***   223.85: INSTR4: Chain up
***   224.05: ACTION: SAY BIRD
***   224.08: Retrieving fact P3: PROPERTY BIRD ANIMAL NIL
***   224.44: INSTR5: Match found
          224.56: All conditions matched
          224.72: Carrying out action AC9: CONST4->AC2
          224.86: Carrying out action AC7: CONST5->AC1
          224.99: All actions done
```




```
***   225.09: ACTION: ANSWER YES
```

If you compare this trace to (test-semantic) without running count first, you will see that quite a few steps have been skipped, even though this is the very first time we run semantic.

A more systematic way of assessing transfer is by to compare latencies of the semantic model with and without running count first. The (do-it n) function performs these runs and prints a table. Here is the result of (do-it 10):

```
Trial Count   Sem-transfer Sem-control
   1     4.8      4.0          5.0
   2     4.7      5.5          6.6
   3     5.1      4.4          4.8
   4     4.8      5.6          6.7
   5     5.2      3.7          4.6
   6     5.0      5.0          6.1
   7     4.5      4.2          4.5
   8     4.8      5.2          5.7
   9     4.7      3.9          4.2
  10     4.0      5.3          5.3
```

This table clearly shows that in the transfer condition case (Sem-transfer) performance is faster than in the control condition (Sem-control). Try different numbers of n to see what difference that makes. Note that the semantic model alternates between two queries that take different amounts of time, therefore the reaction times in the table fluctuate.

Perception and Action

Actransfer uses a simplified version ACT-R's perception and motor modules. This means that some of the precision is lost, but on the other hand that it is easier to program the experiment part of the model. An Actransfer model needs two functions: an initialization function, and a perceptual/motor function.

The initialization function is called every time the (init-task) function is called. This sets up a new trial or block of trials. Like in ACT-R, you have two choices: you either call the function for each trial in the experiment, or you call it for every block of trials. You typically do the former if trials are independent (like in our examples here), but the latter if they are not (in, for example, task switching).

In the most simple case, the init function just sets up the perceptual input, or the initial perceptual input. This is the case in the count example:

```
(defun init-count ()
  (setf *perception* '(two four))
)
```

Whatever you put in the *perception* global variable will be put into the input slots of the model. The semantic function is slightly more complex: it alternates between two types of trials.

The perceptual/motor function is called every time your model puts something in the AC buffer. The action and two optional parameters are passed on to your function. Your function is expected to return a latency (in seconds) of the action. For example, in the count example, the function sets the latency to 0.3 seconds, and gives the reward if the action is "answer":

```
(defun do-action (action &optional h1 h2)
  (when (eq action 'answer) (issue-reward)) ;; give a reward
  0.3)
```

The (issue-reward) function is a predefined function that gives ACT-R the reward you specified in the model specification.

In more complex models, actions will have an effect on perception. For example, the subject presses a key, and as a response the display changes. This is accomplished by changing the

value of the *perception* parameter. As a simple example, in the assignment you will have to make a model that adds two numbers using fingers to track some of the counting. Part of perception is therefore the current number of fingers, and the "add-finger" action adds a finger. We will therefore keep track of the current number of fingers in a global variable *fingers*, and add one to it each time an add-finger action is carried out:

```
(defun do-action-fingers (action &optional h1 h2)
  (when (eq action 'add-finger)
    (incf *fingers*)
    (setf *perception* (list (first *perception*)(second *perception*)(nth *fingers*
'(zero one two three four five six seven eight nine ten)))))
  (when (eq action 'answer) (issue-reward)) ;; give a reward
  0.3)
```

In this example, the first two items in *perception* are the two numbers to be added, and the third is the current number of fingers. An add-finger action increments *fingers*, and updates the third item in the list of *perception* accordingly.

Assignment

The assignment is to add a model of addition by counting to the existing two models, and to assess transfer between counting and addition.

There are two possible solutions to explore. The first is a direct translation of the unit 1 model into Actransfer. The two numbers to be added are perceptual input, and two WM slots are needed to hold the current count and the current sum.

Here is a part of the solution, you'll have to provide the remaining three operators:

```
(add-task add :input (Vaddend1 Vaddend2) :variables (WMcount WMsum) :declarative
((RTcount-fact RTfirst RTsecond))
:pm-function do-action
:init init-add
:reward 10.0
:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -0.5 :alpha 0.2))

(operator :condition (WMcount = nil) :action (zero -> WMcount Vaddend1 -> WMsum
   (say WMcount) -> AC (count-fact WMcount) -> RT) :description "Initialize sum and
   count, retrieve next count")

[... add your own code here ...]

)
```

In the last operator, use the "answer" action to give the answer (as in: (answer WMsum) –> AC)

If your solution is similar to mine, you will find that there is some, but not very much transfer from count to addition (run (do-it-add-single 10) to get this):

```
Trial Count   Add-transfer Add-control
   1     5.0     10.7         11.9
   2     5.0     10.4         11.7
   3     4.9     10.5         10.9
   4     5.1     10.3         10.7
   5     4.5      9.9          9.7
   6     4.5      8.7          9.8
   7     4.5      9.1          9.8
   8     4.6      9.6          9.8
   9     4.4      8.9          9.0
  10     4.0      7.8          9.5
```

The declarative memory graph (on the cover) also shows very little overlap between the two models. Part of the reason is that the model uses two counters, and switches back and forth between them. Apart from not producing much transfer, there is also evidence that people are not so flexible in retaining two counters.

A different solution is to use an external means of counting for one of the counters (i.e., fingers), and do the other one mentally. For this we need the "add-finger" action discussed earlier, and we arrive at a model like:

```
(add-task add-fingers :input (Vaddend1 Vaddend2 Vfingers) :variables
(WMsum) :declarative ((RTcount-fact RTfirst RTsecond))
:pm-function print-action-fingers
:init init-add-fingers
:reward 10.0
:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -0.5 :alpha 0.2))

(operator :condition (Vaddend1<>nil WMsum = nil) :action (Vaddend1 -> WMsum
   (add-finger WMsum) -> AC (count-fact WMsum) -> RT)
   :description "Initialize sum, put up first finger")

[... Add two more operators...]
)
```

Again, you will have to provide the remaining operators. Note that the "(add-finger WMsum)" action actually performs two actions in parallel: it adds a finger, and it says WMsum. The current number of fingers is in Vfingers, and your operators should access that value. If your model is similar to mine, transfer will be better, which you can check using (do-it-add-both *n*). Here is the result of my version of the model, which I ran with n=50 (only the first 10 trials are shown):

Trial	Count	Add-transfer	Add-control	Add-fingers-transfer	Add-fingers-control
1	4.8	9.6	12.0	4.3	7.0
2	5.2	9.6	11.7	4.1	6.5
3	5.0	9.4	11.1	4.9	6.9
4	4.8	9.4	10.7	4.3	6.4
5	4.8	8.8	10.2	3.9	5.8
6	4.8	9.5	10.2	4.5	5.3
7	4.5	8.6	9.2	3.9	5.5
8	4.6	7.7	9.2	3.9	5.6
9	4.4	8.0	9.5	4.5	4.9
10	4.3	6.8	8.9	4.0	5.0