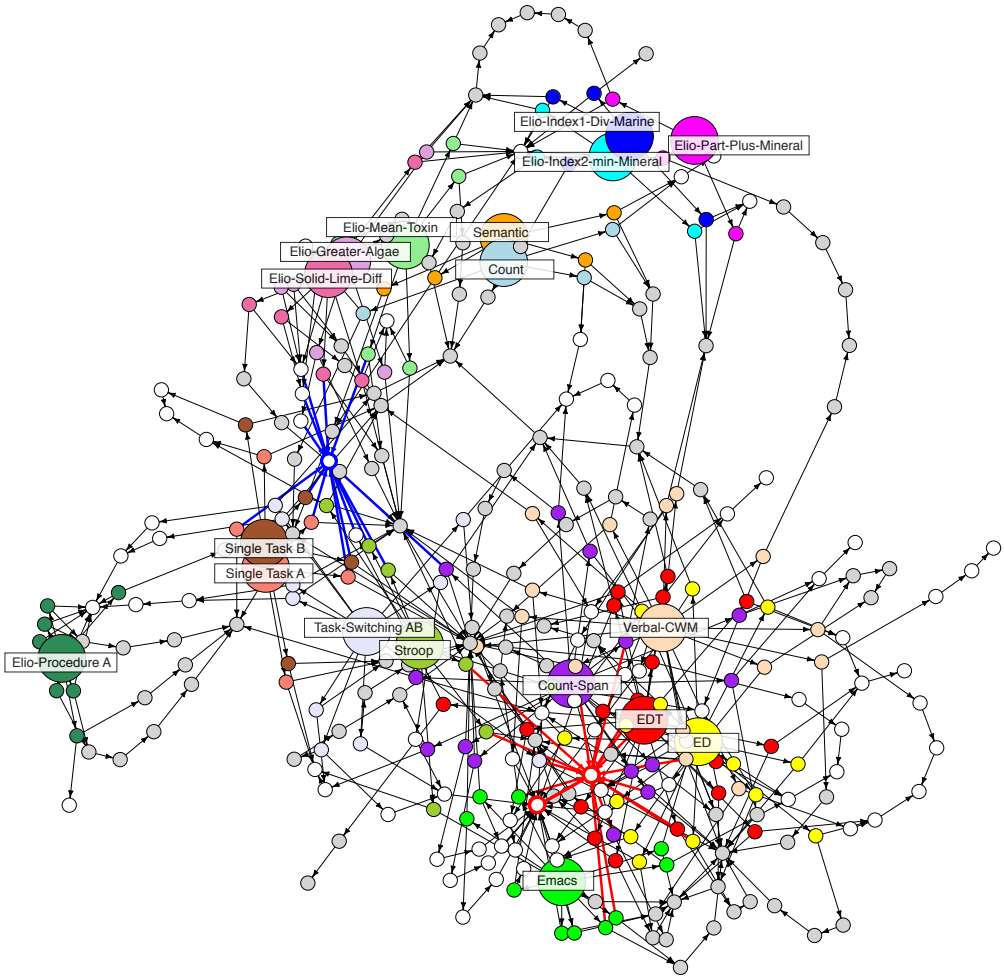
THE NATURE AND TRANSFER OF COGNITIVE SKILLS – SUPPLEMENTAL



Funded by ERC StG
283597 MULTITASK

Niels Taatgen

February 2013

Introduction

This document is an explanation of the Actransfer program that implements the PRIM theory. It will explain the syntax of the models, and will show sample runs of the counting and semantic examples. Instructions on how to run the models are also provided. Actransfer is an extension to the ACT-R architecture, and is loaded on top of ACT-R.

The structure of an Actransfer model

The only thing an Actransfer model needs to provide is a specification of task instructions in ACT-R's declarative memory, and, in most models additional facts in declarative memory (e.g., arithmetic fact). A formal language is provided to specify the operators. In addition to the model code itself, a model file typically contains, as is common in standard ACT-R models, Lisp code to simulate the experiment and to collect data.

The CountSemantic example

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 part of the model is similar to standard ACT-R models. The first items are facts in memory to enable counting. Facts in Actransfer have a fixed structure: they have a name (count0..count5 in the example), followed by "isa fact", and then the contents of the fact prefixed by numbered slots (slot1, slot2, etc.). All facts are enclosed by an (add-dm ...) declaration.

The next part is the declaration of the operators needed for a task. It starts by identifying the goal, and gives labels to slots in the workspace. The workspace consists of a number of buffers, each of which has a number of slots. So, if we talk about a buffer, we refer to a subset of the workspace that is connected to the same cognitive module.

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

A declaration starts with a "(add-instr" followed by the name of the task (count in this case). Subsequently, names are given to each of the slots in the workspace. These names are only for the convenience of the modeler, because they will be anonymized in the eventual model

(they are replaced by generic slots slot1..slotn). The following workspace slots can be declared:

- :input is used for all perceptual input. In this example it has two slots, Vstart and Vend, but there can be more. In the counting task, the input is fixed to the starting and ending number, but in order models the input can and will change depending on the model's actions (or because things just happen in the world).
- :working-memory names slots used for working memory. This model has only one, WMcount.
- :declarative refers to the part of the workspace where facts that are retrieved from memory are placed. It is a list, because there can be multiple different types. In the count example, there are only count-facts.
- There are four control slots that have standard names, so they do not need to be declared. None of these is yet used in the count example:
  - Gtask stores the name of the current task
  - Gcontrol contains the current control state
  - Gtop contains the pointer into declarative memory to refer to the current task context
  - Gparent is used to store a parent goal
- Finally, there are slots for the action. These don't have names.

The next part of the task specification are the following four lines:

```
:pm-function do-action
:init init-count
:reward 3.0
:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -2 :state-activation 0.0 :mas
nil :alpha 0.2))
```

The first 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).

Finally, the three operators needed to count are specified with the following three statements:

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

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

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

These are the three operators specified in Figure 8 of the main article.

They all start with "(ins", 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. On the condition side, there are four types of comparisons that can be made: `Workspaceslot1=Workspaceslot2`, `Workspaceslot1<>Workspaceslot2`, `Workspaceslot1=nil`, or `Workspaceslot1<>nil`. You can use any of the workspace slot names you have defined in these tests, and, in addition, the slotnames Gtask, Gcontrol, Gparent and Gtop. 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. Instead of a Workspaceslot 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 eventual model, constants are put into separate workspace slots, so WMcount=zero will be translated to a comparison between a working-memory slot and a slot in the workspace that holds the constant. A condition involving a Workspaceslot will only be satisfied if that Slot is non-nil, with the exception of `Workspaceslot1=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 `Workspaceslot1—>Workspaceslot2`. In some cases it is convenient to fill a number of slots in one step. For example, instead of `count-fact—>RTcount-fact WMcount—>RTfirst`, you can specify `(count-fact WMcount)—>RT`. 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. Let us look at the first instruction in the counting model.

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

This instruction 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. These specification lists can also contain "?"'s, which means that slot is skipped. For example, `(? WMcount)–>RT` skip filling the first slot in the retrieval, and would therefore retrieve any fact with WMcount in its second slot, not just count-facts.

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

The second and third instruction do the rest of the counting: the second iterates as long as the final number has not been reached, and the third instruction 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 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)`. The function

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

will do all that, and produce the following trace:

```
***     0.34: INSTR0: Initialize Count
***     1.64: ACTION: SAY TWO
***     1.77: Retrieving fact COUNT2: COUNT-FACT TWO THREE NIL
***     2.05: INSTR1: Counting Step
***     3.35: ACTION: SAY THREE
```

```
***     3.49: Retrieving fact COUNT3: COUNT-FACT THREE FOUR NIL
***     3.77: INSTR1: Counting Step
***     5.07: ACTION: SAY FOUR
***     5.11: Retrieving fact COUNT4: COUNT-FACT FOUR FIVE NIL
***     5.48: INSTR2: Finalize count
***     6.46: ACTION: ANSWER FIVE
```

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:

```
***     0.34: INSTR0: Initialize Count
            0.48: Testing condition CD1: V1<>NIL
            0.60: Testing condition CD0: PS1=NIL
            0.73: All conditions matched
            0.90: Carrying out action AC4: V1—>PS1
            1.02: Carrying out action AC3: PS1—>AC2
            1.16: Carrying out action AC2: CONST5—>AC1
            1.31: Carrying out action AC1: PS1—>RT2
            1.43: Carrying out action AC0: CONST6—>RT1
            1.54: All actions done
***     1.64: ACTION: SAY TWO
***     1.78: Retrieving fact COUNT2: COUNT-FACT TWO THREE NIL
***     2.04: INSTR1: Counting Step
            2.16: Testing condition CD3: V2<>RT3
            2.28: Testing condition CD2: PS1=RT2
            2.40: All conditions matched
            2.58: Carrying out action AC5: RT3—>PS1
            2.74: Carrying out action AC3: PS1—>AC2
            2.87: Carrying out action AC2: CONST5—>AC1
            2.98: Carrying out action AC1: PS1—>RT2
            3.11: Carrying out action AC0: CONST6—>RT1
            3.24: All actions done
***     3.40: ACTION: SAY THREE
***     3.42: Retrieving fact COUNT3: COUNT-FACT THREE FOUR NIL
```

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:

```
6.028
6.179
5.792
5.260
5.428
5.449
```

```
5.434
5.076
5.011
4.682
```

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. The goal of the semantic model is to judge relationships between animals and animal categories, and answer question like "Is a canary and 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-instr semantic :input (Vanimal Vcategory) :working-memory
(WMcurrent) :declarative ((RTprop RTitem RTmember-of))
:pm-function do-action
:init init-semantic
:reward 3.0
:parameters ((sgp :lf 0.15 :egs 0.2 :ans 0.1 :rt -2 :alpha 0.2))

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

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

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

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

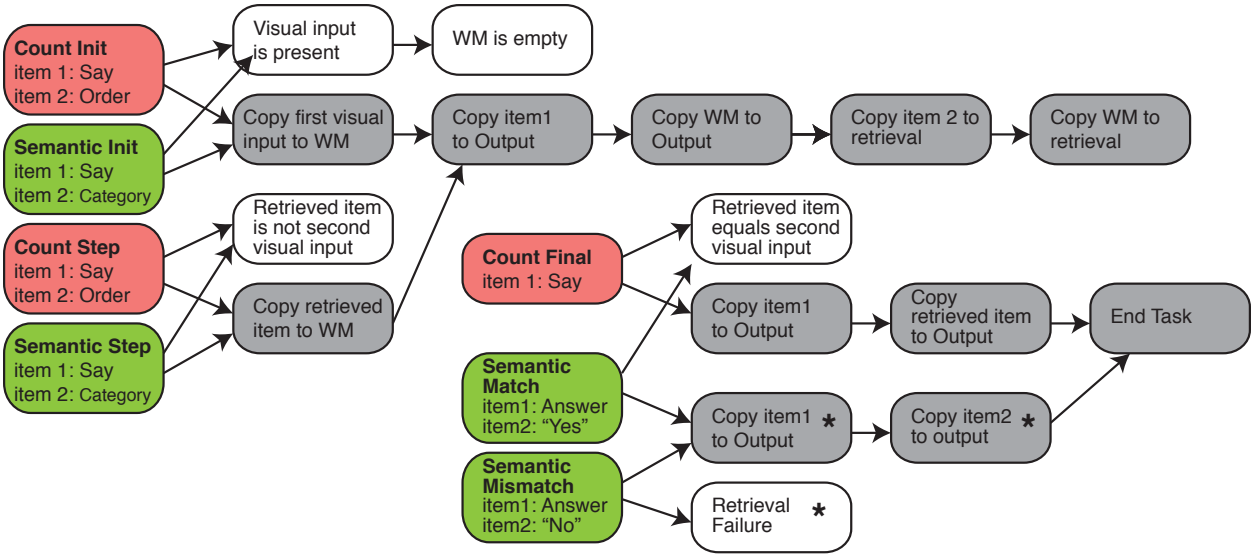
The first two instructions in this model are the same as in the counting model, with the exception that the slot labels are different. But once the instructions are translated into declarative memory, the conditions and actions are identical. The last two instructions are different. If a match between 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 instructions in declarative memory between the two models (Figure 8 in the article, reproduced below).



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     6.6      3.8          4.8
   2     6.4      4.5          6.6
   3     6.4      4.0          5.1
   4     6.3      5.2          6.5
   5     6.3      3.6          4.7
   6     5.9      4.5          6.0
   7     5.7      3.7          4.2
   8     5.4      4.4          5.6
   9     5.3      3.5          4.4
  10     5.2      4.4          5.3
```

This table clearly shows that in the transfer condition case (Sem-transfer) performance is faster than in the control condition (Sem-control). 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 five))
)
```

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 instructions.

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.

The Emacs model

I will discuss the Emacs model in full to give the reader a sense of a complete model (Table 1, below). The model uses four slots to represent perceptual input (Vtype, Vword1 Vword2 and Vline). The perceptual input is used for two purposes. The first is the result of looking at the next edit on the correction instructions in which case Vtype will be set to the type of edit, for example *replace-word*, Vword1 and optionally Vword2 to the words or phrases involved in the edit, and Vline to the line that the edit is in. The second purpose is to represent the result of looking at the screen, in particular at the cursor. In that case Vline

will be set to the line that the cursor is at, and Vword1 to the word that is currently at the cursor.

The model uses one slot in working memory: WMsearch-goal, is a miscellaneous slot that is mainly used to memorize the current search goal (e.g., the line the edit has to be made in, or the word that has to be replaced) so that it can be compared with the current information on the screen. It also uses one control slot, Gcontrol, to track what stage of an edit it is currently in.

Finally, a number of higher-level actions have been predefined for the model. The actions that the model can take are as follows:

- read-instruction: read/interpret the next edit on the correction page. Place the type of edit in Vtype, possible words involved in Vword1 and Vword2, and the line involved in Vline.
- read-screen: look at the cursor position on the screen. Puts the current line in Vline, and the current word in Vword1. Any edit actions also automatically look at the cursor.
- control-n: Emacs command that moves the cursor down one line
- esc-f: Emacs command to move the cursor one word forward
- esc-d: Emacs command to delete the word after the cursor
- control-k: Emacs command to delete the current line
- type-text: Type in the text supplied in the second argument.
- type-text-enter: same as type-text but press enter afterwards.
- next-instruction: move on to the next edit (and read it).

The model's strategy to do a particular edit is relatively straightforward. First, it looks up what the next edit is, and remembers what line the edit is in. It then moves the cursor down until it is at the line in which the edit has to be made. When it is on the right line, there are two options: either it is a word edit or a line edit. If it is a word edit, the model searches for that word and moves the cursor forward word by word until it has found it. Then, depending on whether it is an insert, delete or replace, it deletes the current word and/or inserts a new word. If it is a line edit, it will delete the line and/or type a new line.

The overall structure of the ED and EDT editors is similar to Emacs, even though they are slightly more complex.

Table 1. The operators for Emacs. Bold lines starting with a semicolon are comments.

```
(add-instr emacs :input (Vtype Vword1 Vword2 Vline) :working-memory (WMsearch-goal)
  ;; Initialize: intialize Gcontrol, read the first instruction and determine the first target line
  (ins :condition (Gcontrol=nil) :action ((read-instruction)–>AC  find-goal–>Gcontrol))
  (ins :condition (Gcontrol=find-goal Vtype<>end) :action (Vline–>WMsearch-goal find-line–>Gcontrol
        (read-screen)–>AC))
  ;; Now search for the target line by going down in the page.
  (ins :condition (Vline<>WMsearch-goal Gcontrol=find-line) :action ((control-n)–>AC))
  (ins :condition (Vline=WMsearch-goal Gcontrol=find-line) :action (find-task–>Gcontrol
        (read-instruction)–>AC))
  ;; If the task is something to do with a word, we first find the word by moving the cursor along the line
  (ins :condition (Vtype=replace-word  Gcontrol=find-task) :action (find-word–>Gcontrol
        Vword1–>WMsearch-goal  (read-screen)–>AC))
  (ins :condition (Vtype=delete-word Gcontrol=find-task) :action (find-word–>Gcontrol
        Vword1–>WMsearch-goal  (read-screen)–>AC))
  (ins :condition (Vtype=insert-word Gcontrol=find-task) :action (find-word–>Gcontrol
        Vword1–>WMsearch-goal  (read-screen)–>AC))
  (ins :condition (Vtype=word  Vword1<>WMsearch-goal  Gcontrol=find-word) :action ((esc-f)–>AC))
  (ins :condition (Vword1=WMsearch-goal  Gcontrol=find-word)  :action (word-action–>Gcontrol
        (read-instruction)–>AC))
  ;; Now that we found the word, we are going to do something with it depending on the type of edit
  (ins :condition (Vtype=replace-word Gcontrol=word-action) :action ((esc-d)–>AC  Vword2–>WMsearch-
        goal still-type–>Gcontrol))
  (ins :condition (Vtype=delete-word Gcontrol=word-action) :action ((esc-d)–>AC))
  (ins :condition (Vtype=insert-word Gcontrol=word-action) :action ((type-text Vword2)–>AC))
  (ins :condition (Gcontrol=still-type) :action ((type-text WMsearch-goal)–>AC word-action–>Gcontrol))
  (ins :condition (Vtype=word  Gcontrol=word-action) :action ((next-instruction)–>AC
        find-goal–>Gcontrol))
  ;;; If the task is something with the whole line, carry out the appropriate line command
  (ins :condition (Vtype=delete-line Gcontrol=find-task) :action ((control-k-twice)–>AC
        word-action–>Gcontrol))
  (ins :condition (Vtype=insert-line Gcontrol=find-task) :action ((type-text-enter Vword2)–>AC
        word-action–>Gcontrol))
  (ins :condition (Vtype=replace-line Gcontrol=find-task) :action ((control-k)–>AC
        Vword2–>WMsearch-goal  type-line–>Gcontrol))
  (ins :condition (Gcontrol=type-line) :action ((type-text WMsearch-goal)–>AC
        word-action–>Gcontrol))
  (ins :condition (Vtype=end) :action (finish–>Gtask))
)
```

The Elio model

Table 2 shows the instructions for a part of Procedure A, along with the procedure to calculate the first step in Procedure A. The idea is simple: the slot in the goal that specifies the current task, *Gtask*, is one that can be changed just as the other slots. As soon as the content of that slot is

changed, Actransfer will start retrieving operators for that new task. The model of the Elio task uses this in a very straightforward way: if the model wants to switch to a subtask, it stores its current task in the parent control slot (Gparent), and then puts the name of the subtask in Gtask. The first instruction in Procedure A in Table 2 shows an example of this. Once the subtask is finished, it copies the task name that is stored in Gparent back into Gtask, restoring the main goal. The advantage of this method is that Procedure B can now also use solid-lime-diff, and that any task-specific rules that are learned for solid-lime-diff can be used in both procedures.

Table 2. Part of the instructions for procedure A, and the solid-lime-diff procedure.

(add-instr procedure-a :input (Vlabel Vvalue) :variables (WMatt WMvalue WMprev) :declarative ((RTtype RTarg1 RTarg2 RTans)(RTatt RTvalue RTprev))
;; At the start of the procedure A, save the current task name in Gparent, and put solid-lime-diff in the task name. This switches control to the solid-lime-diff procedure (below). In addition, initialize the declarative memory context by setting Gtop to the current working memory element.
(ins :condition (Gtop=nil) :action (WMid–>Gtop Gtask–>Gparent solid-lime-diff–>Gtask)
;; When control is back to procedure A, the name of calculated result is in WMatt, and the value in WMvalue. WMvalue is typed into the system, after which the next step is initiated (greater-algae).
(ins :condition (WMatt=solid-lime-diff) :action ((enter WMvalue)–>AC [...] greater-algae–>Gtask))
[...four more steps for the remaining calculations])

;; This procedure reads the appropriate information from the screen and performs the calculation solid x ($lime_4$ – $lime_2$)
(add-instr solid-lime-diff :input (Vlabel Vvalue) :variables (WMatt WMvalue WMprev) :declarative ((RTtype RTarg1 RTarg2 RTans)(RTatt RTvalue RTprev))
(ins :condition (Vlabel=nil) :action ((read lime4)–>AC))
(ins :condition (Vlabel=lime4) :action (Vvalue–>WMvalue (read lime2)–>AC))
(ins :condition (RTtype=nil Vlabel=lime2) :action ((subtract WMvalue Vvalue)–>RT (read solid)–>AC))
(ins :condition (RTtype=subtract Vlabel=solid) :action ((mult Vvalue RTans)–>RT))

;; At the end of the calculation, put the result in WMinter, and put the main task back in the task slot in the goal
(ins :condition (RTtype=mult) :action (solid-lime-diff–>WMatt RTans–>WMvalue Gparent–>Gtask)))

The solid-lime-diff subtask is relatively straight-forward. If Vlabel is still nil, this means the model is at the start of the subtask, so it will begin by looking up the value of $lime_4$ on the screen:

```
(ins :condition (Vlabel=nil) :action ( (read lime4)–>AC ))
```

The (read *label*) action looks up the requested value, and puts the label name in Vlabel and the label value in Vvalue. The next operator is retrieved when $lime_4$ has been read:

```
(ins :condition (Vlabel=lime4) :action (Vvalue–>WMvalue (read lime2)–>AC))
```

The action of this operator stores the value of $lime_4$ in WM, and reads $lime_2$. The next operator checks whether $lime_2$ has been read, but also checks one of the retrieval slots. This nil-check on the retrieval is used fairly often before initiating a retrieval. It ensures the same operator is not carried out again after the retrieval is successful.

```
(ins :condition (RTtype=nil   Vlabel=lime2) :action ((subtract WMvalue Vvalue)–>RT
   (read solid)–>AC))
```

The action of this operator is to retrieve a fact with subtract slot1, the value in WM in slot2, and the value in the input in slot3, in other words, it determines the difference between $lime_4$ and $lime_2$. It also reads the value of solid. Once the difference is retrieved and the value of solid is determined, the next operator retrieves the product of the two:

```
(ins :condition (RTtype=subtract  Vlabel=solid ) :action ((mult Vvalue RTans)–>RT))
```

Finally, the result is placed in WM, along with the value name, so that the main goal knows what has been calculated. The final action (`Gparent–>Gtask`) restores the main goal.

```
(ins :condition (RTtype=mult) :action (solid-lime-diff–>WMatt RTans–>WMvalue
   Gparent–>Gtask)))
```

The Chein and Morrison models

The Verbal-CWM models

The article discussed two models of the verbal-CWM task that were part of the Chein and Morrison experiment. I will go over both in detail, because they introduce a number of new concepts.

Table 3 lists the reactive model, which is the more simple of the two. In order to build and maintain a list, there is one control slot, Gtop, which is used to hold the start of the list. The working memory slots in the workspace are used as the end of the list, and one of the WM slots, WMprev, points back to the previous element of the list. In addition to the two declared WM slots (WMconcept, which holds the list item itself, and WMprev, which points to the previous item in the list), there is a third slot called WMid. This is not really a slot, but rather identifies the chunk in WM as a whole. This is used as a reference in the list: the WMprev slot of one element in the list points to the WMid of the previous element in the list. The declarative slots mirror this arrangement with slotnames RTid, RTconcept, and RTprev.

In order to build a list, a first element has to be added. This is achieved by the first operator in the model:

```
(ins :condition (Gtop=nil) :action (WMid–>Gtop))
```

This operator puts the WMid of the current chunk in working memory in the Gtop control slot, which serves as the first element in the list.

Table 3. Reactive model of the Verbal-CWM task.

(add-instr verbal-CWM :input (Vobject Videntity) :working-memory (WMconcept WMprev) :declarative ((RTisword RTlexical RTanswer)(RTconcept RTprev))
;; Initialize the pointer into declarative memory by pointing it at working memory. This will become the first memory item
(ins :condition (Gtop=nil) :action (WMid–>Gtop))
;; Reactive strategy: when there is nothing on the display, wait
(ins :condition (Vobject=pending) :action ((wait)–>AC))
;; Lexical decision task: if there is a word, then retrieve it and answer based on the success of the retrieval
(ins :condition (Vobject=word RT1=nil) :action ((is-word Videntity)–>RT))
(ins :condition (Vobject=word RTanswer=yes) :action ((type "Y")–>AC))
(ins :condition (Vobject=word RT1=error) :action ((type "N")–>AC))
;; If a letter appears, remember it. Put the letter in WM, then bump WM into declarative memory
(ins :condition (Vobject=letter Gtop<>nil) :action (Videntity–>WMconcept (? WMid)–>newWM (wait)–>AC))
;; If the report prompt appears, retrieve and report letters from the memorized list
(ins :condition (Vobject=report RT1=nil) :action (Gtop–>RTid))
(ins :condition (Vobject=report RT1 <> error) :action ((type RTconcept)–>AC RTid–>RTprev))
(ins :condition (Vobject=report RT1=error) :action ((enter)–>AC finish–>Gtask)))

The model then proceeds with lexical decision until a letter appears, which is then put into the list:

```
(ins :condition (Vobject=letter Gtop<>nil)
   :action (Videntity–>WMconcept (? WMid)–>newWM (wait)–>AC))
```

First, the `Videntity–>WMconcept` action puts the letter in working memory. The action after that is one we haven't seen before. It puts new content in the newWM buffer, but this just means that the current contents of WM are cleared (and therefore moved to declarative memory), and to be replaced by the new content. `(? WMid)–>newWM` therefore bumps the current WM chunk into declarative memory, and creates an new chunk with WMid in the second (so WMprev) slot. Effectively, this creates a new list item (see Figure 21 in the main article for an illustration of this process).

When the model receives the report prompt, it has to report all the items that are stored in the list. It starts this process by first retrieving the top first item in the list: `Gtop–>RTid`. In other words, it retrieves a chunk in declarative memory which identifier is in the Gtop slot of the workspace. When the retrieval is successful, in types the letter that is in this slot, `(type RTconcept)–>AC`, and then retrieves the next item in the list, `RTid–>RTprev`.

If the retrieval fails, either because it has successfully reached the end of the list, or because it has forgotten the next item, it presses enter and ends the task.

Table 4. The proactive Verbal-CWM model

```
(add-instr verbal-CWM :input (Vobject Videntity) :working-memory (WMconcept WMprev) :declarative
      ((RTisword RTlexical RTanswer)(RTconcept RTprev))
;; Initialize the pointer into declarative memory by pointing it at working memory. This will become the first
      memory item
(ins :condition (Gcontrol=nil) :action (lexdec–>Gcontrol WMid–>Gtop))
;;; This is just waiting for the first stimulus, otherwise there is no break between lexical decision items
(ins :condition (Vobject=pending Gcontrol=lexdec) :action ((wait)–>AC))
;;; If we are rehearsing and a word appears, switch to lexical decision
(ins :condition (Vobject=word Gcontrol=rehearse) :action (lexdec–>Gcontrol))
;;; Do lexical decision
(ins :condition (Vobject=word RT1=nil Gcontrol=lexdec) :action ( (is-word Videntity)–>RT))
(ins :condition (RTanswer=yes Gcontrol=lexdec) :action ((type "Y")–>AC))
(ins :condition (RT1=error Gcontrol=lexdec) :action ((type "N")–>AC) )
;;; When a letter appears, put it in WM and prepare to rehearse
(ins :condition (Vobject=letter Gcontrol=lexdec) :action (Videntity–>WMconcept (? WMid)–>newWM rehearse–
      >Gcontrol) )
;;; Do rehearsal as long as the letter is still on the screen
(ins :condition (Vobject=letter RT1=nil Gcontrol=rehearse) :action (Gtop–>RTid) )
(ins :condition (Vobject=letter RT1 <> error Gcontrol=rehearse) :action (RTid–>RTprev))
(ins :condition (Vobject=letter RT1=error  Gcontrol=rehearse) :action (Gtop–>RTid))
;;; Do the report
(ins :condition (Vobject=report Gcontrol=rehearse) :action (Gtop–>RTid report–>Gcontrol))
(ins :condition (Vobject=report Gcontrol=lexdec) :action (Gtop–>RTid report–>Gcontrol))
(ins :condition (Vobject=report RT1 <> error Gcontrol=report) :action ((type RTconcept)–>AC RTid–>RTprev))
(ins :condition (Vobject=report RT1=error Gcontrol=report) :action ((enter)–>AC finish–>Gtask)))
```

The proactive model is listed in Table 4. The main difference with the reactive model is the use of a control state (Gcontrol) to track what the model is doing. This is necessary to make sure the model does not confuse a number of situations: if there is a letter on the screen, the model may still need to store it, or is already rehearsing it. If the report prompt is on the screen, the model may be in the middle of reporting, or in the middle of a rehearsal.

Table 5. The model of Stroop.

```
(add-instr stroop :input (Vobject Vcolor Vword) :working-memory (WMconcept) :declarative ((RTmapping
      RTstimulus RTconcept RTstim-type))
;; When the model is waiting for the next stimulus, it can choose to prepare or not
(ins :condition (Vobject=pending  Gcontrol=none) :action (color-task–>Gcontrol (wait)–>AC))
(ins :condition (Vobject=pending) :action ((wait)–>AC))
;; When the stimulus arrives, the model can use its preparation to focus on just color, or it can process both
      properties after all
(ins :condition (Vobject=yes Gcontrol=color-task) :action ((attend-property color-property)–>AC))
(ins :condition (Vobject=yes) :action ((attend-property both)–>AC))
;; If the color is perceived, retrieve the color concept related to the color. This process is slowed down if
      there is a conflicting word in Vword. After the concept is retrieved, say it.
(ins :condition (Vcolor<>nil RTconcept=nil) :action ( (s-mapping Vcolor)–>RT))
(ins :condition (RTconcept<>nil) :action ((say RTconcept)–>AC none–>Gcontrol))
```

To properly prepare, the model has to make the right decision twice. First, it needs to decide to prepare and set Gcontrol to color-task. The alternative is to just wait for the stimulus (this is the competition between the first two operators in the model in Table 5). The second decision is to act upon the preparation once the stimulus does arrive. If the model has prepared, it has the option to specifically attend the color attribute of the stimulus. The alternative is to attend all properties of the stimulus. If the model made the "right" decision twice (a 25% probability without prior training), it will suffer from less interference on a Stroop conflict-trial. The two rules that lead to less interference both have a condition that checks visual input (Vobject) in combination with checking the control state (Gcontrol). This particular combination is also present in the Verbal-CWM model that rehearses (in fact, four rules have that condition), but not in the model that does not rehearse. By training that rule during the Verbal-CWM task, a production is learned through production compilation that matches both internal and external state to particular values (similar to the learned-condition-rule in Table 5), so it is just one extra step to learn the task-specific rule for the Stroop task.

The Karback and Kray models

The principle behind the models of the Karbach and Kray experiment is the same as the Chein and Morrison experiment. The Stroop model is identical to the model used in the Chain and Morrison simulation. The countspan task is different from the verbal-CWM model in that the task is different, but the rehearsal principle is the same. The only difference is that the countspan task also has the option to not rehearse, so at the moment

that rehearsal would normally be initiate, it has an alternative operator that decides to just wait.

In the control condition the model is training on a simple choice reaction task. The model is in Table 6. This model is quit straightforward, and does not require a control state.

Table 6. Model of a simple choice reaction task (in this case the food task)

```
(add-instr single-task-A :input (Vobject Vfood Vsize)  :declarative ((RTmapping RTcategory RTkey))
;;; Wait for the stimulus
(ins :condition (Vobject=pending) :action ((wait)–>AC))
;;; If there is a stimulus, get its food property
(ins :condition (Vobject=yes) :action ((get-property food-property)–>AC))
;;; Retrieve the key associated with the property
(ins :condition (Vfood <> nil RTkey=nil) :action ((mapping Vfood)–>RT))
;;; Press the key
(ins :condition (RTkey <> nil) :action ((press-key RTkey)–>AC))
(ins :condition (Vobject=last) :action (finish–>Gtask))
```

Table 7 shows the task switching model. It is fairly elaborate because it implements both a reactive and a proactive strategy. The reactive strategy waits for the next stimulus to appear before determining the task, while the proactive strategy accomplishes this during the interstimulus interval.

Installing and running Actransfer

The "HowToRun" document that is supplied with the models describes how Actransfer can be run, and also outlines how the results of the model can be plotted using the supplied R scripts.

Table 7. Task switching model for the food/size task

```
(add-instr task-switching-AB :input (Vobject Vfood Vsize) :variables (WMcur-task WMcount) :declarative
        ((RTmapping RTcategory RTkey)(RTother RTfirst RTsecond))
;;; Initialize model, set first task to the food-task, set a counter to one
(ins :condition (WMcur-task=nil) :action (food-task–>WMcur-task  one–>WMcount do-task–>Gcontrol
        (wait)–>AC))
;;; Get the property appropriate for the current task
(ins :condition (Gcontrol=do-task Vobject=yes WMcur-task=food-task )
        :action ((get-property food-property)–>AC))
(ins :condition (Gcontrol=do-task Vobject=yes WMcur-task=size-task)
        :action  ((get-property size-property)–>AC))
;;; Get the key that needs to be pressed and press it
(ins :condition (Vfood <> nil RTkey=nil) :action ((mapping Vfood)–>RT))
(ins :condition (Vsize <> nil RTkey=nil) :action ((mapping Vsize)–>RT))
(ins :condition (Gcontrol=do-task RTkey <> nil) :action ((press-key RTkey)–>AC))
;;; The following operators give a choice between proactive and reactive control.
;;; Proactive is to determine the the next task in the inter stimulus interval
(ins :condition (Vobject=pending  Gcontrol=do-task) :action (choose-task–>Gcontrol))
;;; Reactive is to wait for the next stimulus before determining the next task
(ins :condition (Vobject=pending  Gcontrol <> choose-task) :action (choose-task–>Gcontrol (wait)–>AC))
;;; Update of the task: if the count is one, just set it to two
(ins :condition (WMcount=one Gcontrol=choose-task) :action (do-task–>Gcontrol two–>WMcount
        (wait)–>AC))
;;; If the count is two, reset it to one, and retrieve what the other task is.
(ins :condition (RTsecond=nil WMcount=two Gcontrol=choose-task ) :action ((other-task WMcur-task)–>RT))
(ins :condition (RTsecond <> nil Gcontrol=choose-task ) :action (do-task–>Gcontrol one–>WMcount
        RTsecond–>WMcur-task (wait)–>AC))
(ins :condition (Vobject=last) :action (finish–>Gtask))
```