02 – Environment, motor and vision module - the demo model

August 30, 2016

1 The demo model

The second documented model comes from unit2 of tutorials in Lisp ACT-R.

In [1]:
```
import string
import random
import warnings

import pyactr as actr


class Environment(actr.Environment): #subclass Environment
    """
    Environment, putting a random letter on screen.
    """

    def __init__(self):
        self.text = string.ascii_uppercase
        self.run_time = 2

    def environment_process(self, start_time):
        """
        Environment process. Random letter appears,
        model has to press the key corresponding to the letter.
        """
        time = start_time
        yield self.Event(time, self._ENV, "STARTING ENVIRONMENT")
        letter = random.sample(self.text, 1)[0]
        self.output(letter, trigger=letter) #output on environment
        time = time + self.run_time
        yield self.Event(time, self._ENV, "PRINTED LETTER %s" % letter)


environ = Environment()

m = actr.ACTRModel(environment=environ)

g = m.goal("g")
g2 = m.goal("g2", set_delay=0.2)
actr.chunktype("chunk", "value")
actr.chunktype("read", "state")
actr.chunktype("image", "img")
actr.makechunk(nameofchunk="start", typename="chunk", value="start")
```

```
actr.makechunk(nameofchunk="attend_let", typename="chunk", value="attend_let")
actr.makechunk(nameofchunk="response", typename="chunk", value="response")
actr.makechunk(nameofchunk="done", typename="chunk", value="done")
g.add(actr.chunkstring(name="reading", string="""
        isa     read
        state   start"""))

t1 = m.productionstring(name="find_unattended_letter", string="""
        =g>
        isa     read
        state   start
        ?visual>
        state   free
        ==>
        =g>
        isa     read
        state   attend_let
        +visual>""")

t2 = m.productionstring(name="encode_letter", string="""
        =g>
        isa     read
        state   attend_let
        =visual>
        isa     _visual
        object  =letter
        ==>
        =g>
        isa     read
        state   response
        +g2>
        isa     image
        img     =letter""")

m.productionstring(name="respond", string="""
        =g>
        isa     read
        state   response
        =g2>
        isa     image
        img     =letter
        ?manual>
        state   free
        ==>
        =g>
        isa     read
        state   done
        +manual>
        isa     _manual
        cmd     'presskey'
        key     =letter""")

sim = m.simulation(realtime=True, environment_process=environ.environment_process,\
                   start_time=0)
```

```
        sim.run(2)
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: find_unattended_letter')
OUTPUT ON SCREEN
G
END OF OUTPUT
(0.05, 'PROCEDURAL', 'RULE FIRED: find_unattended_letter')
(0.05, 'g', 'MODIFIED')
(0.05, 'visual', 'CLEARED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'visual', 'ATTENDED TO OBJECT')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: encode_letter')
(0.15, 'PROCEDURAL', 'RULE FIRED: encode_letter')
(0.15, 'g', 'MODIFIED')
(0.15, 'g2', 'CLEARED')
(0.15, 'visual', 'CLEARED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.35, 'g2', 'CREATED A CHUNK: image(img=G)')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: respond')
(0.4, 'PROCEDURAL', 'RULE FIRED: respond')
(0.4, 'g', 'MODIFIED')
(0.4, 'manual', 'COMMAND: presskey')
(0.4, 'g2', 'CLEARED')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.65, 'manual', 'PREPARATION COMPLETE')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'NO RULE FOUND')
(0.7, 'manual', 'INITIATION COMPLETE')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'NO RULE FOUND')
(0.8, 'manual', 'KEY PRESSED: G')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'NO RULE FOUND')
(0.9, 'manual', 'MOVEMENT FINISHED')
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9, 'PROCEDURAL', 'NO RULE FOUND')
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:89: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

Compared to the first model (discussed in 01), we are now adding environment and we let our ACT-R model interact with the environment (using vision and motor modules).

2 ACT-R model

We focus on vision and motor module here. Notice that these modules are not instantiated in our ACT-R model. That's because pyactr does that automatically and it reserves the names "manual" and "visual" for these two modules in production rules. It also reserves chunk types _manual and _visual that are the only possible chunk types that could be used in these modules.

2.1 Vision module

Vision module allows the information that appears in environment to enter ACT-R models.

There are two most common processes for vision. You can query it, using "?". In particular, you can query whether its buffer is empty or not, or you can query its state ("free" or "busy").

Aside from querying, you can also request something into the visual buffer, using "+". Unlike with the retrieval buffer, you don't specify any values for vision (see the request "+visual>" in the rule find_unattended_letter). It will put into the value of "object" of its chunk whatever is present in environment.

The vision module is very simple in this respect. It doesn't care about any details of what is to be seen, it just puts it all in. In this respect, it is currently more primitive than LispACTR. Modifications to this will follow.

2.2 Motor module

Motor module can mainly be queried ("?") or a chunk can be requested, using "+".

2.2.1 Requesting in motor module ("+")

Unlike other modules, motor module does not put anything into its buffer. Requesting simply means that the motor module will carry out the given command (right now, only pressing a key is possible).

This has one consequence. You cannot request an arbitrary chunk in the motor module. You can only request a special chunk. The chunk is called "_manual" and it has two slots. "cmd" (for command) specifies what should be done - currently, only key pressing is possible. "key" specifies what key should be pressed. In other words, this requests that "a" is pressed:

```
In [2]: "+manual>  isa _manual cmd 'presskey' key='a'"

Out[2]: "+manual>  isa _manual cmd 'presskey' key='a'"
```

2.2.2 Querying in motor module ("?")

You can query the state of the motor module. But motor module has a few more states. According to ACT-R, carrying out a motor command has subphases: preparation phase, processor and execution. You can query any of these states in motor module. Here is a table from the Lisp ACT-R reference manual, summarizing the states:

Preparation state	Processor state	Execution state	When
FREE	FREE	FREE	Before event arrives
BUSY	BUSY	FREE	When event is received
FREE	BUSY	BUSY	After preparation of movement
FREE	FREE	BUSY	After initiation movement
FREE	FREE	FREE	When movement is complete

Motor module can be interrupted in preparation state by a new request. After that, it will carry out its process and it will not be interrupted, so it might make sense to query for preparation state.

Currently, the implemented motor module is very primitive. It does not actually calculate time to carry out a process, it just takes default values from Lisp ACT-R.

2.3 Imaginal buffer

Imaginal buffer is a late comer to ACT-R. It is like goal, in that it specifies the overarching aim in the task. But unlike goal, it takes time to set. Commonly, the set time is 0.2 seconds. There is no separate module for imaginal buffer in pyactr. You simply create it by creating a second goal buffer, with set_delay=0.2.

```
In [3]: example = actr.ACTRModel()
        imaginal = example.goal("imaginal", set_delay=0.2)
        print(imaginal)
```

```
set()
```

2.4 Chunks

In contrast to 01 – Introduction to symbolic system – the count model, this model does not define chunks using chunkstring, but makechunk. In makechunk, attribute value pairs are keyword arguments of the function, and you can define the name of the chunk and the type of the chunk(both arguments are optional). The name of the chunk has to be filled in if you are going to refer back to the chunk in other chunks or in rules, i.e., if it is a value of some slot (as it is the case here).

The following two notations are equivalent.

```
In [4]: actr.makechunk(typename="chunk", value=10)
```

```
Out[4]: chunk(value=10)
```

```
In [5]: actr.chunkstring(string="isa chunk value 10")
```

```
Out[5]: chunk(value=10)
```

3 Environment

Repeated here:

```
In [6]: class Environment(actr.Environment): #subclass Environment
            """
            Environment, putting a random letter on screen.
            """

            def __init__(self):
                self.text = string.ascii_uppercase
                self.run_time = 2

            def environment_process(self, start_time):
                """
                Environment process. Random letter appears, model has to press the key corresponding to
                """
                time = start_time
                yield self.Event(time, self._ENV, "STARTING ENVIRONMENT")
                letter = random.sample(self.text, 1)[0]
                self.output(letter, trigger=letter) #output on environment
                time = time + self.run_time
                yield self.Event(time, self._ENV, "PRINTED LETTER %s" % letter)
```

The environment is a class, subclassing Environment present in the package. When we initalize it, we specify how long the environment will be running in "run_time".The default value is 1s.

The environment class should have a method which will specify an environment process. In this case, the process consists in (i) selecting a letter, (ii) printing on the screen.

Printing on the screen is taken care of by the superclass. You call the method "output" and specify what should be printed and what trigger should the environment have. The trigger is the crucial interaction with the ACT-R model. It says to what action the environment will respond. In this case, a random letter is printed on screen and environment will respond to pressing whatever was printed on the screen. The response means that the current output is closed and the environment process moves to the next step.

Environment also specifies events taking place in it (self.Event). You have to have the first event to start your environment. The third slot is the name of the action, the second slot is the name of the process. Normally, it's good to use self._ENV for the second slot it's just the name "ENVIRONMENT". It doesn't matter what names you use for these two slots. The only crucial part is the first slot, which specifies at what time the event should take place. Most likely, the first event will take place when the whole simulation starts, as it is here.

```
In [7]: example = environ.Event(time=0, proc=environ._ENV, action="STARTING ENVIRONMENT")
```

The second event would take place after the amount of time given in the first argument (in this case, it is 10s). However, if trigger is done meanwhile, the second event quietly disappears and the environment process moves on to the next step (in this case, there is none).

Some more examples of environments are in the folder examples_environment.

4 Running the model

The model and the environment are initialized:

```
In [8]: environ = Environment()
        m = actr.ACTRModel(environment=environ)
```

Note that the model now specifies one argument, environment. This lets the ACT-R model know that there is an environment it can interact with.

After that, we added all necessary bits and pieces and started the simulation of the model.

```
In [9]: dm = m.DecMem()
        g = m.goal("g", default_harvest=dm)
        g2 = m.goal("g2", default_harvest=dm, set_delay=0.2)
        g.add(actr.chunkstring(name="reading", string="""
                isa     read
                state   start"""))

        t1 = m.productionstring(name="find_unattended_letter", string="""
                =g>
                isa     read
                state   start
                ?visual>
                state   free
                ==>
                =g>
                isa     read
                state   attend_let
                +visual>""")

        t2 = m.productionstring(name="encode_letter", string="""
                =g>
                isa     read
                state   attend_let
                =visual>
                isa     _visual
                object  =letter
                ==>
                =g>
                isa     read
                state   response
```

```
        +g2>
        isa     image
        img     =letter""")

m.productionstring(name="respond", string="""
        =g>
        isa     read
        state   response
        =g2>
        isa     image
        img     =letter
        ?manual>
        state   free
        ==>
        =g>
        isa     read
        state   done
        +manual>
        isa     _manual
        cmd     'presskey'
        key     =letter""")


sim = m.simulation(realtime=True, environment_process=environ.environment_process,\
                   start_time=0)
```

A few new things are present in the last variable assignment. First, we state realtime as True. This ensures that the simulation will be run in real time.

The next bit specifies what process we will use in our environment. We use "environment_process", which consists in (i) printing a random letter on screen, (ii) waiting for the model to press the same key (see above). After that we have to supply keyword arguments that the selected environment process takes (apart from self).

The next step is running the simulation.

```
In [10]: sim.run(2)
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: find_unattended_letter')
OUTPUT ON SCREEN
X
END OF OUTPUT
(0.05, 'PROCEDURAL', 'RULE FIRED: find_unattended_letter')
(0.05, 'g', 'MODIFIED')
(0.05, 'visual', 'CLEARED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'visual', 'ATTENDED TO OBJECT')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: encode_letter')
(0.15, 'PROCEDURAL', 'RULE FIRED: encode_letter')
(0.15, 'g', 'MODIFIED')
(0.15, 'g2', 'CLEARED')
(0.15, 'visual', 'CLEARED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
```

```
(0.35, 'g2', 'CREATED A CHUNK: image(img=X)')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: respond')
(0.4, 'PROCEDURAL', 'RULE FIRED: respond')
(0.4, 'g', 'MODIFIED')
(0.4, 'manual', 'COMMAND: presskey')
(0.4, 'g2', 'CLEARED')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.65, 'manual', 'PREPARATION COMPLETE')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'NO RULE FOUND')
(0.7, 'manual', 'INITIATION COMPLETE')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'NO RULE FOUND')
(0.8, 'manual', 'KEY PRESSED: X')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'NO RULE FOUND')
(0.9, 'manual', 'MOVEMENT FINISHED')
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9, 'PROCEDURAL', 'NO RULE FOUND')

/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:89: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

And we can check various buffers or declarative memory (added here).

```
In [11]: print(g2)

set()

In [12]: print(g)

{read(state=chunk(value=done))}

In [13]: print(dm)

{image(img=X): {0.4}, _visual(object=X): {0.15}}
```

Notice that the declarative memory is not empty! Why not? Because the visual and g2 module were cleared at some point in our rules. And every module, when cleared, sends chunks off into the declarative memory it is associated with. The memory is specified by default_harvest in case of goal and visual buffers and by declarative_memory in case of retrieval buffers. If there is only one declarative memory, it is enough to specify it once, other buffers will be associated with it automatically.

Notice that the declarative memory also specifies when buffer clearing happened. E.g., we see that the memory received a chunk at 0.15 and 0.4s.