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

March 29, 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 pyactr.environment as env
import pyactr.model as model

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

    def __init__(self):
        self.text = string.ascii_uppercase
        self.run_time = 10

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

class Model(object):
    """
    Model interacting with environment -- pressing the same key as the letter that appears on t
    """

    def __init__(self, env):
        self.m = model.ACTRModel(environment=env)

        self.dm = self.m.DecMem()

        retrieval = self.m.dmBuffer("retrieval", self.dm)

        g = self.m.goal("g", )
```

```
                g2 = self.m.goal("g2", set_delay=0.2)
                self.start = self.m.Chunk("chunk", value="start")
                self.attend_let = self.m.Chunk("chunk", value="attend_let")
                self.response = self.m.Chunk("chunk", value="response")
                self.done = self.m.Chunk("chunk", value="done")
                g.add(self.m.Chunk("read", state=self.start))

            def find_unattended_letter(self):
                yield {"=g": self.m.Chunk("read", state=self.start), "?visual": {"state": "free"}}
                yield {"=g": self.m.Chunk("read", state=self.attend_let), "+visual": None}

            def encode_letter(self):
                yield {"=g": self.m.Chunk("read", state=self.attend_let), "=visual": self.m.Chunk("_visu
                yield {"=g": self.m.Chunk("read", state=self.response), "+g2": self.m.Chunk("image", ent

            def respond(self):
                yield {"=g": self.m.Chunk("read", state=self.response), "=g2": self.m.Chunk("image", ent
                yield {"=g": self.m.Chunk("read", state=self.done), "+manual": self.m.Chunk("_manual",

        environ = Environment()
        m = Model(environ)
        m.m.productions(m.find_unattended_letter, m.encode_letter, m.respond)
        sim = m.m.simulation(realtime=True, environment_process=environ.environment_process, start_time=
        sim.run(1.5)
        print(m.dm)
```

```
Warning: Chunk type chunk not defined; added automatically
Warning: Chunk type read not defined; added automatically
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
Warning: Chunk type image not defined; added automatically
(0, 'PROCEDURAL', 'RULE SELECTED: find_unattended_letter')
OUTPUT ON SCREEN
P
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(entity=P)')
(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: P')
(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')
{_visual(object=P): {0.15}, image(entity=P): {0.4}}
```

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.

2.1 Vision module

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

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 basically put into its buffer 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 a 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, reserved in the class Chunk for this module. 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 [10]: {"+manual": m.m.Chunk("_manual", cmd="presskey", key="a")}
```

```
Out[10]: {'+manual': _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 and to understand it you should realize one thing: 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.

3 Environment

Repeated here:

```
In [2]: class Environment(env.Environment): #subclass Environment
            """
            Environment, putting a random letter on screen.
            """

            def __init__(self):
                self.text = string.ascii_uppercase
                self.run_time = 10

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

The environment is a class, subclassing Environment present in the package. When we initalize it, we specify for how 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. You have to have the first event to start your environment. Its action (the third slot) could be anything, its process (the second slot) could be anything. (env._ENV will conveniently name the process "ENVIRONMENT"). 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 [3]: example = env.Event(time=0, proc=env._ENV, action="STARTING ENVIRONMENT")
```

(Note that in the actual code, we are using the function "roundtime" inside Event. This rounds time to miliseconds.

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.

3.1 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 [4]: example = model.ACTRModel()
        imaginal = example.goal("imaginal", set_delay=0.2)
```

4 Running the model

The model and the environment are initialized:

```
In [5]: environ = Environment()
        m = Model(environ)
```

Note that the model now takes one variable, environment, which is passed onto ACTRModel (initialized as self.m in the model). This lets the ACT-R model know that there is an environment it can interact with.

The next step is adding production rules to the ACT-R model.

```
In [6]: m.m.productions(m.find_unattended_letter, m.encode_letter, m.respond)
```

After that, we start simulation.

```
In [7]: sim = m.m.simulation(realtime=True, environment_process=environ.environment_process, start_time=
```

A few new things are present here. First, we state realtime as True. This ensures that the simulation will be run in real time. This is not necessary.

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

The next step is running the simulation.

```
In [8]: sim.run(1.5)
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: find_unattended_letter')
OUTPUT ON SCREEN
Z
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(entity=Z)')
(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: Z')
(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')
```

We let it run for 1.5 s. (Even though, clearly, 1s would be enough.)
Finally, we check declarative memory.

```
In [9]: print(m.dm)
```

```
{_visual(object=Z): {0.15}, image(entity=Z): {0.4}}
```

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