Comment to Source Code 'vw4.py'

Gerd Doeben-Henisch
Email: gerd@doeben-henisch.de

August 19, 2019

1 The Problem and the Solution Vision

In this post there are two goals: (i) The one is the continuation of learning to use the programming language *python*, the other (ii) is the usage of the Actor-Actor Interaction (AAI) paradigm to describe a problem and its solution.

In the AAI paradigm (see the page `https://www.uffmm.org/2019/05/12/aci-frontpage/`) it is assumed that one describe a vision solution by a sequence of states connected by changes. A *state* is a collection of *facts* which describe *objects* with properties and *relations* between these objects. It is possible to introduce additionally *actor models* which describe possible *inner states (IS)* of those objects which are *actors*. Actor objects have *inputs (I)* from the environment, *outputs (O)* to the environment and *inner states (IS)* which define a *behavior function* ϕ.

2 The Actor-Actor Stories (AASs)

For this post it has been assumed that there exists a *virtual world (VW)* which can *interact* with a *user (U)* in some way. Based on this communication the virtual world will organized its *internals states* in a way that there are data structures for a *2-dimensional grid* with *empty spaces* ('_'), which can be occupied either by *obstacles ('O')*, *food objects ('F')*, or by *actors ('A')*.

This configuration allows the distinction between two different Actor-Actor stories depending from two different story levels. At *actor actor story level 0 (AAS L0)* we have the real user interaction with an interface for the virtual world. At *AAS L-1* we are inside the virtual world where the actor objects are interacting with the other objects in the 2-dimensional grid world. In every AAS we can find certain specific changes.

Figure 1 shows a simplified graphical description of two embedded actor-actor stories. The AAS L0 has basically states with the user as executive actor *USER(eA1)* and the virtual world *VW(aA1)* as an assisting actor. Both are embedded in a communication relation *COM(eA1, aA1)* which allows different kinds of inputs and outputs from each side.

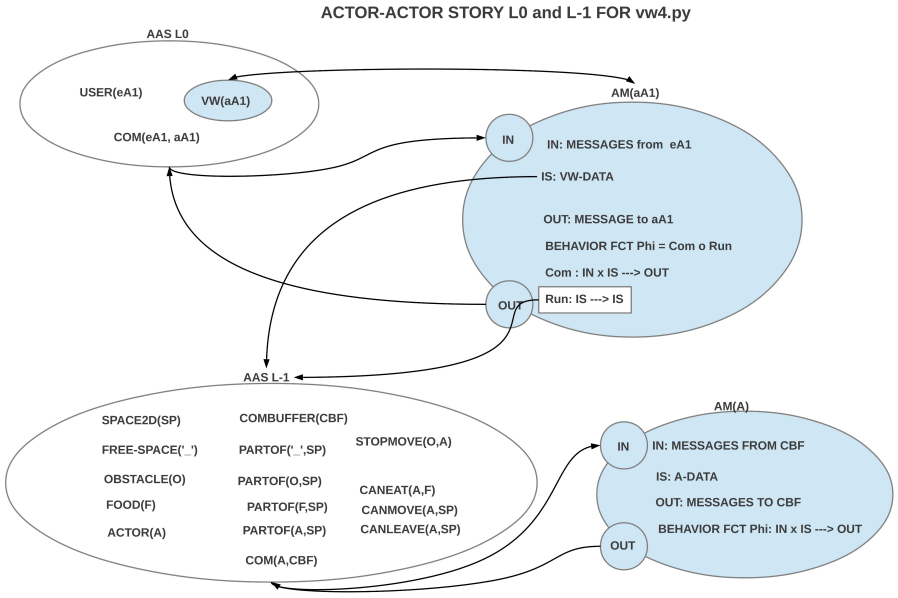

Figure 1: Overview of two actor-actor stories at level 0 and level -1

3 The Actor Models (AMs)

To understand the interactions between the real user USER(eA1) and the virtual world as actor VW(aA1) one needs at one hand some *understanding* of the real user, and some understanding of the virtual world as actor. In case of the virtual world this can be enabled by an explicit *actor model (AM)* of the virtual world as actor. This can be done by describing which kinds of *inputs* the virtual world actor can receive from the real user and which kinds of outputs the virtual world actor can produce for the real user. Additionally one can describe the internal states of the virtual world actor. In case of the internal states we can recognize a 2-dimensional space model organized as a *n*n grid* whose cells can be occupied by obstacles, food, and actors. While the food objects ('F') can only change their energy level, the actor objects ('A') can *move* around, can *eat* and they will thereby change their energy level, either *increasing* by eating food or by *decreasing* moving around.

On level -1 the virtual world *manages* the changes by receiving input from the actors for intended eating and moving and has to decide, whether these actions are possible and if yes, what kind of changes this implies.

In the actual code the explicit communication between virtual world actor objects and the virtual world manager is not yet explicitly coded. This eill be extended in the upcoming posts.

4 The Python Code

The source code can be found on this page: `https://www.uffmm.org/2019/08/19/starting-with-python3-the-very-beginning-part-9/`

Two different files can be distinguished: the main program *vw4.py* and many additional functions collected in the import file *vwmanager.py*. This second file will be automatically imported by the file vw4.py during run time.

4.1 Main Program vw4.py

The main program asks the user some questions about some parameters which will determine the layout of the virtual world.

1. The object *objL* contains some global properties for the *food objects* and the *actor objects* of the virtual world. In that moment, when concrete food and actor objects will be activated in the 2-dimensional grid of the virtual world these global properties will be added to these objects.
2. The object *say* indicates whether the program shall show the user a maximal amount of information about the process or a minimal amount. A maximal amount will usually only be interesting during the development of the program.
3. The object *m* encodes the n umber of rows and columns of the 2D-grid. The rows are representing the y-axis from above (=0) to below (m-1). The columns representing the x-axis from left (=0) to right (m-1).
4. The object *mx* represents the 2D-grid.
5. The *olO*, the *olF* and the *olA* objects represent respectively the lists of the *obstacles*, the list of the *food objects*, as well as the list of the *actor objects* as they have been randomly be assigned to the 2D grid.
6. The *percentage of objects* ask in the beginning is at this point only approximately, because all the asked objects will be randomly be distributed and the later generated objects can overwrite those, which are already attached to a cell in the 2D grid. Therefore a *final computation* of the real distribution will be shown at the end of the object generation phase.
7. Another important object is the *CYC* object which represents the number of wanted cycles. Then a *world clock WCLCK* will be started to count the number of cycles which have been run so far. This clock can be used to inform the user about the *time of the virtual world*.
8. With the objects *stepMode* and *cont* the user can decide whether he can follow the run step by step or continuously without interruptions.
9. With the expression *len(olA)* one asks for the actual length of the list of all actor objects. If an actor object looses too much energy then it can 'die' and then it will be removed from the 2D-grid. Thus if the length of

the oIA list is below 1 then all actor objects have been removed and the simulation will be stopped although the wanted number of cycles is not yet fulfilled.

10. During the *main loop* different actions are activated and processed to enable the movement of all actor objects, their eating, and the update of the energy levels of all objects. As last action of all these it will be checked whether an actor objects has to be removed from the grid on account of a possible energy level below 1.