YAES simulator: a tutorial

Lotzi Bölöni

April 2, 2008 - version 0.2

Contents

1 The big picture

YAES is a time-step simulator, that is, the simulator performs something at every timestep.

The input to the simulators are:

- the input parameters (`SimulationInput`).
- the simulation code (this is a class implementing the `yaes.framework.simulation.SimulationCode` interface).
- what to do before simulation starts (in the `setup` function), at every timestep (in the `update` function) and after the simulation finishes.

During running the simulation, the current data of the simulation is carried in the *context*, a class implementing the `yaes.framework.simulation.context` interface. It is important that you keep all the information in this class, rather than in other classes you write - this allows many cool things like re-running only parts of a series of simulation, processing the results of the simulation later, distributing a simulation over many machines, and so on.

Finally, the output of the simulation is captured in the `yaes.framework.simulation.SimulationOutput` class. You don't need to overwrite this class, it is fine as it is. For your convenience, the `SimulationOutput` class will carry all the input parameters, measurements you might have made, and the final version the of context.

`SimulationOutput` is serializable, and if you kept the context serializable as well, then you can just write and load it to a file and you have the results of the simulation stored for future inspection.

2 How to run a simulation?

We will assume that you have the input parameters, the context class `MyContext`, and the simulation code class `MyCode`.

In the first approximation, there are two ways to run a simulation. In the interactive mode, you step through the simulation step by step. This makes sense during debugging or demoing. Nevertheless, you might want to have some sort of visualization as at least informative text output happening in this case.

Alternatively, you just want to run a simulation as fast as possible and you are only interested on the output. This is important in many scientific studies where you need to run the same simulation many times and get the average values, or, alternatively, you need to run the simulation many times while varying them or more parameters.

2.1 Running a simulation interactively

This is what you will put somewhere in your code:

```
SimulationInput si = new SimulationInput();
si.setStepTime(100); //run for 100 cycles
si.setParameter("Temperature", 70 );
```

In practice, almost always you will define a constant for the string "Temperature". Now for the run itself:

```
MyContext context = new MyContext();
simulationOutput = SimulationControlGui.simulation(si, MyCode.class, context);
```

And this is it. Note that the simulation code is passed as a class, rather than an external instantiation.

If you run this code, it brings up the window in Figure 1.

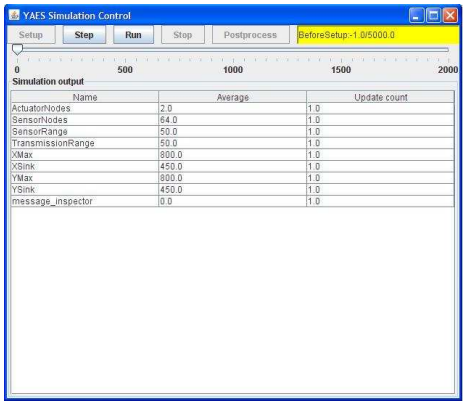

Figure 1: screen shot for controlling the simulation

You first need to push Setup, then you can choose Step or Run. If you are in the middle of a run, you can stop it at the next timestep by pushing Stop (you can't stop in the middle of a timestep).

The table under the buttons show the variables of the simulation output. Note that all the parameters of the *SimulationOutput* were already copied to the output.

2.2 Running a simulation without interactive control

To run a simulation without interactive control you call:

```
out = Simulation.simulate(si, MyCode.class, Context);
```

The GUI will not appear, and the simulation will run automatically from beginning to end (which is usually the maximum number of cycles specified in the simulation input). In the following we discuss in more detail the components of the simulation.

3 The components of a simulation

3.1 The simulation input

The simulation input class is a simple repository of parameters addressed by name. It can store integers, doubles, strings and enums. Only that when you read them, you need to know the type:

```
SimulationInput si = new SimulationInput();
si.setParameter("Temperature", 70);
double temp = si.getParameterDouble("Temperature");
```

The simulation input can be created with an optional parameter taking another simulation input object. The effect is to copy all the parameters over to the new input. This comes in handy if you need to generate inputs where only one parameter differs (parameter sweep).

3.2 The simulation output

The simulation output is also a repository of values addressed by name. However, its main focus is the processing and manipulating of the statistic properties of the values. Let us see an example. Create a simulation output and update its variable `"X"`:

```
SimulationOutput so = new SimulationOutput();
so.update("X", 1.0);
so.update("X", 5.0);
so.update("X", 4.0);
```

Now we can inspect this variable. We can, of course, retrieve its last value:

```
double x = so.getValue("X", RondomVariable.Probe.LASTVALUE);
```

which will be, of course, 4. But we can retrieve the maximum, minimum and average:

```
xmax = so.getValue("X", RandomVariable.Probe.MAX);
xmin = so.getValue("X", RandomVariable.Probe.MIN);
xaverage = so.getValue("X", RandomVariable.Probe.AVERAGE);
```

We can retrieve the sum:

```
xsum = so.getValue("X", RandomVariable.Probe.SUM);
```

the number of times the variable was updated:

```
xcount = so.getValue("X", RandomVariable.Probe.COUNT);
```

or the variance:

```
xvariance = so.getValue("X", RandomVariable.Probe.VARIANCE);
```

We can also retrieve the lower and higher range of the 95% confidence interval.

All these come in handy in the post simulation analysis, the generation of the graphs and so on.

3.3 The simulation code

The simulation code class needs to implement the actual simulation. It appears that this will be a very complex class but in practice it is usually very simple.

One thing to remember: you should keep the SimulationCode class **stateless**, that is, do not create any variables here. All the state of the simulation should go into the context.

Here is what you would have in the setup:

```
public void setup(SimulationInput sip, SimulationOutput sop,
    IContext theContext) {
    final DirectedDiffusionContext context =
         (DirectedDiffusionContext) theContext;
    context.initialize(sip, sop);
}
```

So basically you initiate the context. The context, having state is worth initializing. The code, not having any variables, does not have anything with initializing.

The `postprocess()` function gives you an opportunity to do some calculations after the simulation had been finished. One example would be to calculate values which can not be calculated during simulation. In the first approximation, leave this function empty, and add code on the need-by-need basis.

3.4 The update function

This is where the real work of the simulation happens. At every timestep, this function needs to perform all the activities which advance the simulation such as:

- move the vehicles
- perform the message
- agents make decision
- energy gets consumed
- and so on

However, this functionality is the business of the individual objects which are part of the environment. These objects need to provide an "update" function which perform the work. The responsibility of the update function in the simulation code is to call the update function of all the relevant objects in the context.

This update function takes as a parameter the current time, and return an integer value. Return 1 if you want to continue the simulation for another timestep, return 0 if you want to terminate it early (the simulation will terminate anyhow when the specified timestep in the simulation input expire).

3.5 The context

As we said previously, the context is the repository of the current state of the simulation. There are essentially two different type of objects which you keep here:

- constants and variables which are here for convenience as measurement purposes
- active objects, which need to be updated (more exactly, be themselves update) at every simulation step. Examples are agent, vehicles, network nodes, sensors and so on. One special object of this type is the World.

The **World** object is supposed to represent the environment in which the active objects operate. There are two ready-mode world objects in YAES:

- `yaes.framework.world.World`: a generic world for embodied agents. Contains time, a map, a list of named locations, a directory of objects.
- `yaes.framework.world.sensornetwork.SensorNetworkWorld` in addition to the regular world, it maintains a list of sensors, actuators (mobile node such as intruder). It is also managing the communication and perception among the nodes.

In an ideal world, you need to develop a world model for your specific application. In a lesser world, just take one of the existing ones, and if it does not cover what you need, complement it with objects in your context.

4 How to create a video?

We assume that you have a running simulation of your chosen application. We assume that you have a working display of your simulation on the visual panel.

The first step is to convince YAES to save an image of the visual panel at every simulation step. To do this, you need to go to your `xxxSimulation.java` class (the one which implement `ISimulationCode`). In the `update` function you will probably find the call to repaint the visual display. Right after that, you need to enter the saving of the file, so the result should be as follows:

```
context.getVisual().repaint();
String fileName = String.format("video%03d.jpg", (int) time);
context.getVisual().saveImage(fileName);
```

This will save it in files `video001.jpg, video002.jpg`... etc. What remains is to create a video out of these files. You can use your favorite video editor program - some versions of Microsoft Windows come with a simple one.

Another way is to use `mencoder`. Download the `mplayer` package from `http://www.mplayerhq.hu`. The command line you want to use is something like this:

```
mencoder mf://*.jpg -mf w=800:h=600:fps=25:type=jpg -ovc
     lavc -lavcopts vcodec=mpeg4:mbd=2:trell -oac copy -o output.avi
```

For more complete information, check `http://www.mplayerhq.hu/DOCS/HTML/en/menc-feat-enc-images.html`.

5 Generating graphs and presenting results

5.1 Parameter sweeps

We assume that you have your simulation running, and that you are collection the performance results of your simulation in the `SimulationOutput`. Now, you want to prepare a presentation or a paper with your performance results. The accepted was to do this is to show a graph, which contains on the X axis something which is considered an input parameter (such as number of nodes) and on the Y axis something which is considered a performance metric (such as packets lost).

Let us consider first the data acquisition. The normal way to do this is to perform the simulation for all the values of the specific input parameter in the range, while keeping all the other parameters constant. This process is called a *parameter sweep*. Parameter sweeps tend to be computationally expensive, thus you don't want to run them visually - much more important than the visualization is the ability to see how far they have progressed and how much is left.

This is how you perform a parameter sweep:

```
List<SimulationInput> inputs = new ArrayList<SimulationInput>();
for (int nodes = 10; nodes < 100; nodes = nodes + 5) {
    SimulationInput sim = new SimulationInput(model);
    sim.setParameter("Nodes", nodes);
    inputs.add(sim);
}
List<SimulationOutput> outputs =
   Simulation.simulationSet(inputs, MySimulation.class, MyContext.class);
```

To put it simple, you create a list of `SimulationInput` objects which differ in only one parameter, feed it to YAES, and what you get is an array of `SimulationOutput` object. One little thing to consider is that the context will be created with a default constructor - thus if you need to initialize it, do it in the `setup()` function of the simulation.

Many times you want to compare your algorithm against its competitors (eg. random, greedy or differently parametrized versions of the same algorithm). You will need to repeat the parameter sweep for each competitor. The output of this process is a collection of lists of simulation output objects.

5.1.1 Saving your data

The lists of SimulationOutputs are serializable and can be read and written to a file using the save, saveList, restore and restoreList functions in SimulationOutput.

Use them. Do not generate a long function which does all the calculations in memory and at the end generates the graphs and discards the data. Put the generation of the graphs and the running of simulation in separate menu items.

You might want to check the `Simulation.cachedSimulationSet()` function, which allows you to re-run only parts of the simulation.

Although it seems a wasted time, the effort put in to run your simulation efficiently, is time well spend. You will waste more time re-running the simulation over and over.

5.2 Generating graphs

We assume that you have the lists of simulation outputs as discussed above. Now, you can generate a graph by specifying for each individual line on the graph

- what list of simulatoon outputs to take the data from
- what value to put on the X axis - this will be the parameter you have done the parameter sweep on
- what value to put on the Y axis - this would be the performance metric of interest, and its appropriate statistical sample: last value, average, minimum, maximum and so on.

Here is an example:

```
List<SimulationOutput> myalgorithm... \\ obtained from simulation
List<SimulationOutput> randomalgorithm... \\ obtained from simulation

PlotDescription pd = new PlotDescription("The number of nodes",
   "The performance");
pd.addPlotLine(new PlotLineDescription("Nodes", "Performance",
"My algorithm", myalgorithm));
pd.addPlotLine(new PlotLineDescription("Nodes", "Performance",
       "Random algorithm", randomalgorithm));
pd.generate(new File("performancegraph.m"));
```

Note that a different constructor of `PlotLineDescription` class allows you to plot various aggregates such as maximum, minimum, average, sum and so on.

This segment of code will create a file `performancegraph.m`. You need to simply run this file in Matlab to display the graph. You can use the Matlab tools to customize the graph to your liking and then save it in the desired format, such as EPS for inclusion in LaTeX.