AESOP-ACP Howto

Gian Paolo Jesi

April 2, 2014

This document represents a gentle introduction to AESOP-ACP framework programming. AESOP-ACP is the result of merging two pieces of software: Agents and Emergencies for Simulating Organizations in Python (AESOP) and Agents and Complexity in Python (ACP). This document provides the basic knowledge required to configure and use the simulator and to customize agent classes.

AESOP-ACP has been developend as an evolutionary step of the java Enterpise Simulator - jES (`http://web.econ.unito.it/terna/jes/`) - developed by Pietro Terna.

1 Introduction

This tutorial is a step by step guide to use, configure and customize the simulator. The document uses the 'embroidery' example provided in the AESOP-ACP distribution, where an embroidery plant is simulated.

It is supposed that you have access to:

- basic knowledge of the Python language and basic programming in general is welcome
- the Python language installed on your machine (version >= 2.7.x)[1]
- the following Python packages are required for running AESOP-ACP:
  - networkx (`http://networkx.github.io/`)
  - matplotlib (`http://matplotlib.org/`)
  - numpy (`www.numpy.org`)
  - xlrd (`https://pypi.python.org/pypi/xlrd`)
  - pyparsing (`http://pyparsing.wikispaces.com/`)

[1] It should work on 2.6.x as well, but it is not tested.

- the aesop-acp package (`http://sourceforge.net/projects/aesop-acp/`) installed on your system and linked by the PYTHON_PATH

This tutorial IS NOT exhaustive at all, but it is enough to get you started. The goal is to give the reader the basics of AESOP-ACP and a step-by-step guide to understand the 'embroidery' example provided.

2 Simulator introduction

2.1 Why AESOP-ACP

Simulation is a valuable tool in which a synthetic model of the object we are interested in is generated and it can be analysed and hypothesis can be tested before having deployed anything in the real world.

AESOP-ACP has been developed to generate hypothesis and test them quickly in a dynamic environment where each element is wired in an evolving network. The aim of our framework is focused on production process modelling and supply chains, however, it is basically suited in Complex Adaptive Systems scenarios. In particular, when the network of relations among the actors is a key factor.

AESOP-ACP has been heavily inspired by the jES Enterprise Simulator and it is supposed to be its next step evolution especially in terms of usability and flexibility. In addition, Swarm-Like Agent Protocol in Python (SLAPP) and especially the AESOP-SLAPP software can be also considered an ancestor of the current AESOP-ACP. However, while written in Python it is still related to the Swarm `http://www.swarm.org/` library scheme. The jES simulator, SLAPP and AESOP-SLAPP has been developed by Pietro Terna.

The combination of Agent Based Modelling (ABM) and Python language provide a high level abstraction when designing the problem and great reconfigurability and simplified access to endless third party libraries and extensions.

AESOP-ACP has been designed with a pluggable component architecture in mind. Essentially, each key feature (e.g., scheduling parsers, agent's selection policy, graphical interfaces, ...) is managed by a default specific class which can be substituted by a more suitable one from the command line or configuration file. This design is the main contribution to its flexibility.

2.2 AESOP-ACP gentle and brief overview

AESOP-ACP is conceptually split in two main functional components: (i) the ACP core, which provides the actual (event-based) simulation features and network abstractions and (ii) AESOP which adds flexible reconfiguration options and *recipes*: a simple grammar that defines statements suitable to express "what to do" in a production processes.

The underlying simulation model is fully asynchronous and event based. Being an ABM flavoured software, the entities that populates the virtual environment are called *agents*. Each agent must be an instance of the *Agent* class or its sub-class.

An agent reacts to received *events* or *messages* according to its *passive behaviour*. A specific event or a time interval may trigger the *active behaviour*, which mimics the 'conscious behaviour' of the simulated entity. By default, only the agent's passive behaviour is enabled.

The simulation produces observation data through *Monitor* class objects. Basic functionalities, such as the agent's queue load, are monitored by default. AESOP-ACP simulations can be run in text console mode or in graphical mode[2] to provide the user with a rapid, visual feedback achieved by real-time graphical plots of the model behaviour or performance.

A simulation in AESOP-ACP first need a configuration.

A basic production process simulation can be carried out by just providing a correct configuration.

A configuration has to provide the following information:

- who are the agents involved
- how each agent is configured
  - **optional**:provide a "who knows whom" relation, or how the agents are wired in the network [3]
- what to do
- in which sequence

The simplest way to provide a configuration is trough a pair of files whose default names are: '`agent_graph.dot`' and '`schedule.xls`'.

The former provides the agent object definitions and their parameters. The file format follows the GraphViz (`http://www.graphviz.org/`) graph format. Essentially, it defines a graph whose nodes are agents of a specific class type and having particular attributes expressed by a key/value pair. In addition, the graph edges can be specified. If no edges are present, the simulator considers a fully connected topology.

The latter instead, schedules and triggers actions or events. We adopted a spreadsheet format[4] to lower the burden of our framework, since non-technical user feels comfortable with this kind of tools.

In the specific production process scenario we focus on, the schedule sets when orders are placed, with which frequency and how the orders are made.

Orders are specified using *recipes*. Briefly, a recipe is a list of production steps or actions required to full-fill the order. Each step states the abilities required to accomplish

[2] Using the `-G` or `--gui` switch options from the command line.

[3] When no network is provided, the simulator considers a fully connected graph: all agents are free to communicate with anyone. Providing another specific graph instead, means giving a (possibly strong) constraint over the system, while we are usually interested in letting the simulator find the best arrangement for routing orders in a self emergent fashion. In other words, we are more interested in the emergent communication pattern that arises from the agent behaviour.

[4] MS Excel 97 format, which can be easily managed by third-party libraries.

the current task and the time it takes. Preconditions required for a step or boolean relations (i.e., 'and', 'or' statements) among steps can be expressed as well. The concept of a recipe will be discussed in the following sections in more detail.

The schedule file can handle many other features that are not discussed in this document, such as: agent group management, fine grain definition of the simulation loop and other details.

2.3 AESOP-ACP life-cycle

The life-cycle of the a simulation is as follows. The agents graph file is read first and a graph object having the network representation is generated. The nodes of the graph represents the agents that are created as well by a factory method using the parameters found. There are no mandatory parameters for agents since if nothing is specified, an instance of *aesop.agents.core_agents.Agent* class is adopted. In practice, we have to specify the class we need (e.g., *aesop.agents.PRAgents.Unit*) with the `classType` parameter.

Then the simulator reads the schedule file and stores information about the orders structure and their scheduling over time. Events are generated to inject the orders into the system at the required time. In production process simulation, usually the orders have a specific entry point in the system. Basically, each order is delivered to a specific agent (e.g., *aesop.agents.PRAgents.OrderDistiller*) which is responsible of managing the order and to route it to the first working station according to the virtual factory policy.

The next step is to initialize the actors involved in the virtual environment. This can be done by calling specific initialization methods from the schedule file. However, for simple simulation, this is not required, since object initialization from the parameters in the agent graph file is sufficient.

The simulation engine is then started. Events are popped one by one from a global queue and delivered to the destination agent. When an agent receives the event, it immediately process the event if having the resources (e.g., not busy) or it leaves the event in its local queue for further process in the next future.

If the active behavior is enabled[5], agents can take their own action at time-triggered intervals and are not limited to react to external stimuli.

If the graphical interface has been selected, the GUI will appear and the actual simulation will start as soon as the button 'START' is pressed.

The simulation runs until the global queue becomes empty or a specific end time is specified by the switch '`-u`', '`--until`' from the command line or from the global parameter '`until`' in the graph file[6].

[5]The user has to provide an implementation for the agent `handleBehavior()` method, which is empty in production process agent.

[6]The one defined in the graph file has the priority over the one of the command line.

3 A practical example

Let's start with a practical usage of AESOP-ACP through an example. We run the simulator by typing the following from the command line:

```
> python statsim.py -h

usage: startsim.py [-h] [-g FILE] [-d DIR] [-s FILE] [-m SCHEDULEMGR_CLASS]
                   [-l LOG_INTERVAL] [-u DURATION_TIME] [-c CLASS_GUI] [-G]
                   [-D] [-P SELECTPOLICYMGR_CLASS] [-M] [-v]

Command Line Interface to AESOP-ACP simulation framework. By default,
it reads the 'agents_graph.dot' in the current directory, if any.

optional arguments:
  -h, --help            show this help message and exit
  -g FILE, --gFile FILE
                        specify the graph file to read from. Default is
                        'agents_graph.dot' in project directory
  -d DIR, --projectDir DIR
                        project directory: where is the graph file. Requires
                        final '/' char
  -s FILE, --scheduleFile FILE
                        schedule file: a .xls file with the scheduled
                        activities (optional)
  -m SCHEDULEMGR_CLASS, --scheduleMgr SCHEDULEMGR_CLASS
                        schedule manager class
  -l LOG_INTERVAL, --logtime LOG_INTERVAL
                        how often the time is printed
  -u DURATION_TIME, --until DURATION_TIME
                        time duration of the simulation
  -c CLASS_GUI, --classgui CLASS_GUI
                        class handling the GUI (only if GUI is active, e.g.,
                        -G or --gui). Default is 'aesop.gui.simgui.simGUI'
  -G, --gui             toggle GUI generation
  -D, --dynamic_network
                        considers the network dynamic for gui updates
  -P SELECTPOLICYMGR_CLASS, --selectPolicyMgr SELECTPOLICYMGR_CLASS
                        policy manager class when choosing the next agent
  -M, --monitor_output  toggle data save for each monitor of each agent (can
                        be overridden by specifying
                        'enable_monitor_output=True' in an agent entry in
                        agentsGraph file
  -v                    toggle verbosity

>
```

AESOP-ACP prints the in line help showing all the valid options.

We refer to the 'embroidery' example distributed with the simulator. It is available in the folder: "/aesop/examples/embroidery".

3.1 Inside the agent_graph file

We first give a look to the content of the 'agents_graph.dot' file:

```
1 strict digraph  {
2   center=True;
3   size="8x11";
```

```
4     title="Agents Graph";
5     //until=100; // overrides previous definitions
6
7     office    [classType="aesop.agents.PRagents.OrderDistiller",
8
9  tags="stdembroidery1t,laser\_embroidery1t,press\_strass\_embroidery1t,
       paillettes\_embroidery1t,manual\_strass\_embroidery1t"
10    ];
11    machine1   [classType="aesop.agents.PRagents.Unit",
12      capacity=4,
13      tags="embroidery,lasercut"
14    ];
15    machine2   [classType="aesop.agents.PRagents.Unit",
16      capacity=1,
17      tags="embroidery,lasercut,paillettes"
18    ];
19    cutter1  [classType="aesop.agents.PRagents.Unit",
20      capacity=1,
21      tags="papercut,alucut"
22    ];
23    cutter2  [classType="aesop.agents.PRagents.Unit",
24      capacity=1,
25      tags="papercut,alucut"
26    ];
27    /* cutter3   [classType="aesop.agents.PRagents.Unit",
28      capacity=1,
29      tags="papercut,alucut"
30    ]; */
31    /* cutter4   [classType="aesop.agents.PRagents.Unit",
32      capacity=1,
33      tags="papercut,alucut"
34    ]; */
35    /* cutter5   [classType="aesop.agents.PRagents.Unit",
36      capacity=1,
37      tags="papercut,alucut"
38    ]; */
39    /* cutter6   [classType="aesop.agents.PRagents.Unit",
40      capacity=1,
41      tags="papercut,alucut"
42    ]; */
43    /* cutter7   [classType="aesop.agents.PRagents.Unit",
44        capacity=1,
45        tags="papercut,alucut"
46      ]; */
47    cleaner1   [classType="aesop.agents.PRagents.Unit",
48      capacity=1,
49      tags="clean"
50    ];
51    /* cleaner2 [classType="aesop.agents.PRagents.Unit",
52      capacity=1,
53      tags="clean"
54    ]; */
55    /* cleaner3 [classType="aesop.agents.PRagents.Unit",
56      capacity=1,
```

```
57      tags="clean"
58    ]; */
59    press [classType="aesop.agents.PRagents.Unit",
60      capacity=1,
61      tags="peperstrass,cleanpaper"
62    ];
63    strass  [classType="aesop.agents.PRagents.Unit",
64      capacity=1,
65      tags="putstrass"
66    ];
67
68    /* Buffers: */
69    buffer  [classType="aesop.agents.PRagents.Warehouse",
70      capacity="unbounded",
71      tags="store",
72      start\_full="True",
73      shape=rectangle
74    ];
75
76    /* Agent wiring: */
77    // office -> machine1;
78    // machine1 -> office;
79    // office -> machine2;
80    // machine2 -> office;
81    // office -> cutter;
82    // cutter -> office;
83    //  office -> cleaner;
84    // cleaner -> office;
85    // office -> press;
86    // press -> office;
87    // office -> strass;
88    // strass -> office;
89
90    //  machine1 -> buffer;
91    // buffer -> machine1;
92    // machine2 -> buffer;
93    // buffer -> machine2;
94    /* machine2 -> cleaner; */
95    // cutter -> buffer;
96    // buffer -> cutter;
97    // cleaner -> buffer;
98    // buffer -> cleaner;
99    // press -> buffer;
100   // buffer -> press;
101   // strass -> buffer;
102   // buffer -> strass;
103 }
```

It is a human readable ASCII file, having a C-like syntax. The syntax is the GraphViz one which is extended (or mixed) with AESOP-ACP specific parameters.

As suggested in **line 1**, a directed graph is defined. **Lines 2-5** are dedicated to global definitions such as the title and other details. Note that the simulator specific '`until`' parameter is commented out, therefore the simulation will end when no more events are

lying in the global queue.

Lines 7-9 are dedicated to the definition of the first graph node, which is - of course - an agent. "Office" is the node/agent name and the squared parenthesis hold its definition. The '`classType`' parameter links to the fully qualified name of the class from which the simulator factory will forge the object instance. Office agent is an *OrderDistiller* object. '`tags`' is a very important parameter for AESOP-ACP. It is a list of one or more strings representing the distinct tasks that an agent is capable to perform. Note that the office agent can perform tasks that correspond to order names, this fact is fundamental being the office the simulation entry point, but we will discuss more about this point later in these sections.

Lines 10-13 define an agent called "machine1" of type *Unit*, which is capable of both 'embroidery' and 'lasercutting' (i.e., `tags` parameter). In addition, machine1 has a `capacity`[7] equal to 4, which means that can manage to run 4 tasks simultaneously.

All other agent definitions are quite similar except for "buffer" agent at lines 40-45. It is a *Warehouse* type having an unbounded `capacity`. In a Warehouse agent the `capacity` corresponds to the storage space and 'store' is the default tag. When a *Warehouse* is generated, its storage is empty and in simple simulation setups managing the supplies can complicate quite a lot the simulation. In this case, setting 'True' the `start_full` parameter solves the issue.

Starting from **line 47** the code regarding the graph wiring is commented out. This implies that the simulator automatically considers a fully connected network. In other words, any agent is free to communicate with any other agent.

3.2 Inside the schedule file

The schedule file is a spreadsheet which is organized in several sheets. The main one is called 'schedule'. Other reserved sheet names are the following: 'recipes', 'simcontrol' and 'groups'. This document just focuses on schedule and recipes sheets.

The following is the content of the recipes sheet in 'embr_schedule_sec.xls':

	A	B	C	D
1	#	1		
2	macro	production	every=400	until=1200
3	#	2		
4	#	macro	production	

A line starting with symbol '#' followed by number x in the next cell means that next lines applies to time x. For example, **line 1** means that what will follow are action to be scheduled at time 1.

Line 2 defines a *macro* named 'production'. A macro is a set of actions which are defined in a separated sheet which is named as the macro itself. The scheduling modifiers '`every`' and '`until`' can re-run the macro at regular intervals. In this particular case,

[7] If not stated otherwise, the standard `capacity` value is 1.

the production macro will fire at time 1 and then every 400 time units until time 1200. When not specified otherwise[8], numbers correspond to the smallest time unit considered by AESOP-ACP which is a second. This macro will be called at time 1.

What it is supposed to do at time 2 is commented out.

The following is the content of the 'production' macro sheet.

	A	B	C	D
1	stdembroidery	doOrder	stdembroidery	1
2	laser_embroidery	doOrder	laser_embroidery	1
3	stdembroidery	doOrder	stdembroidery	2
4	press_strass_embroidery	doOrder	press_strass_embroidery	10
5	stdembroidery	doOrder	stdembroidery1t	2
6	paillettes_embroidery	doOrder	paillettes_embroidery	8
7	manual_strass_embroidery	doOrder	manual_strass_embroidery	2

Essentially, each line corresponds to a method invocation. Consider **line 1**: method '*doOrder()*' is invoked at time 1 over an agent that can perform a 'stdembroidery' task. The simulator finds a suitable agent by looking at the agents tags. In our current setup, the office agent is capable of doing that and implements also the required method. The rule for method invocation in the schedule file is:

$$\texttt{who}, \texttt{method}, \texttt{param}_1, \texttt{param}_2, \ldots, \texttt{param}_\texttt{n}$$

The `who` field can be a tag as in the current case or can be a specific agent name or class name. All the agents that match the statement will receive the call event.

The next columns contains the method parameters in the correct order: the order name and how many orders must be generated. The simulator is going to schedule an event (*aesop.acp.ScheduleEvent*) that will trigger this method invocation over the office object at time 1.

This is the basic mechanism that injects the orders into the agent environment.

Essentially order names correspond to the abilities that the *OrderDistiller* agent (e.g., the office agent in our example) has.

After receiving the event corresponding to the *doOrder()* invocation, the *OrderDistiller* generates the order object and routes it to the first processing agent.

The policy with which an agent chooses the nest agent to route the order to is given by a *SelectPolicyManager* class. The default policy is to pick the agent with the smallest queue available (i.e., see 'selectpolicies.py' module) and many others can be designed from scratch.

But, what an agent it is supposed to do when we receive an order? This aspect is managed by the recipes, in the 'recipes' sheet. We focus on **line 3** regarding the

[8]Using the symbols 's' (second), 'm' (minute), 'h' (hour), 'd' (day) after the number. For example: every=2h.

recipe for 'stdembroidery'. Note that every element or symbol of a recipe fills a single spreadsheet cell. Every processing step is separated by a comma, with the exception of 'and' and 'or' statements[9]. The following is the recipe for 'stdembroidery' order in text form:

```
stdembroidery p papercut 1 paper buffer, papercut s 10, embroidery s 10,
                                clean s 30
```

The recipe starts with its (unique) name. This represents the id with which it will be addressed in the simulation.

```
p papercut 1 paper buffer
```

Symbol 'p' stands for *procurement* step: it is a special statement where we express the need of procuring the goods required to accomplish the task we are going to do. In a procurement it must be stated who is actually making the procurement, how many pairs of symbols expressing what to get and where to get and the actual pairs. In other words, this step requires that an agent capable of 'papercut' (e.g., implementing papercut and thus having 'papercut' symbol in its tag list) procures a single good which is a 'paper' unit from an agent called 'buffer'. Being a procurement, the agent to get the goods from must be a *Warehouse* agent.

```
papercut s 10
```

Next step is made of just three elements and it is a *basic* step. Essentially, a basic step defines how long a task it will take.

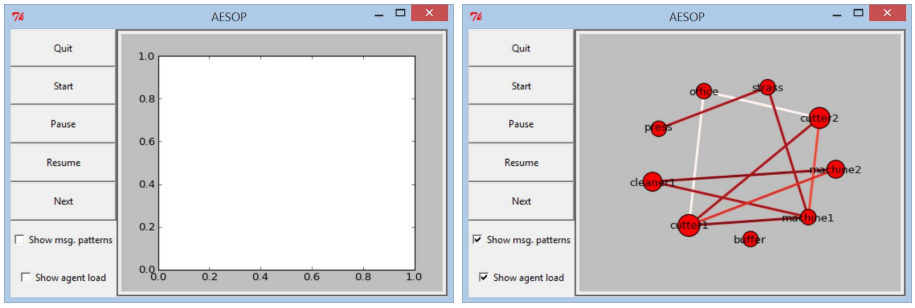
The first symbol 'papercut' tells that the agent running this step must be capable of 'papercut'. Being this step after a procurement, the actual agent performing this step will be the same as before. The 'papercut' task is going to involve the agent for 10 seconds: symbol 's' in the recipe stands for seconds (i.e., the symbols 'm', 'h', 'd' meaning respectively minutes, hours and days are valid as well) and number 10 is the actual amount of time units required.

```
embroidery s 10
```

Another basic step that takes 10 seconds to be accomplished, but requires an agent with 'embroidery' abilities.

```
clean s 30
```

[9] In these cases, the steps are separated by the parenthesis surrounding the elements involved in the 'and' or 'or' statement. Inside the elements surrounded by the parenthesis multiple steps are separated with commas.



(a) Waiting for input (b) While running

Figure 1: Basic GUI in AESOP-ACP framework. The left picture (a) shows the basic GUI when just started and waiting for input, while the right picture (b) is a snapshot of a run.

Final recipe step which is again a basic one. It takes 30 seconds and requires an agent with 'clean' abilities.

Being the last step of a recipe, this step has something extra to care about. Its outcome - some kind of good - must be stored somewhere or it will lost! When a basic step is the last element of a recipe, the agent processing it stores the produced good in its local storage buffer.

In a real scenario, each recipe should end with 'end' steps. These steps tell agents where to store their goods (i.e., in which *Warehouse* agent) and to express other details such as: a specific name to store the good and express the time taken for the storage operation.

For a deeper discussion about recipe steps and recipe grammar the reader should check the AESOP-ACP reference manual.

3.3 Running the example

In order to run the example, just type the following[10]:

```
> python startsim.py -d aesop/examples/embroidery/ -s rm_schedule_sec.xls
-G
```

The `startsim.py` is a script interface that runs the simulations. The '`-G`' switch triggers the generation of the GUI depicted in Figure 1. On the left of the window the five main buttons are dedicated to the simulation run and the frame on the right side is dedicated to show the network structure that wires the agent if any. Using the 2 switches on the bottom left, it is possible to respectively emphasize the main routes where the orders (messages) are exchanged and the queue load on each agent.

First click on both the switches to engage the plot of message patterns and queue agent loads. By clicking the 'start' button the actual simulation starts. Figure 1(b) shows the

[10] Note that according to your actual machine OS, the path syntax may change.

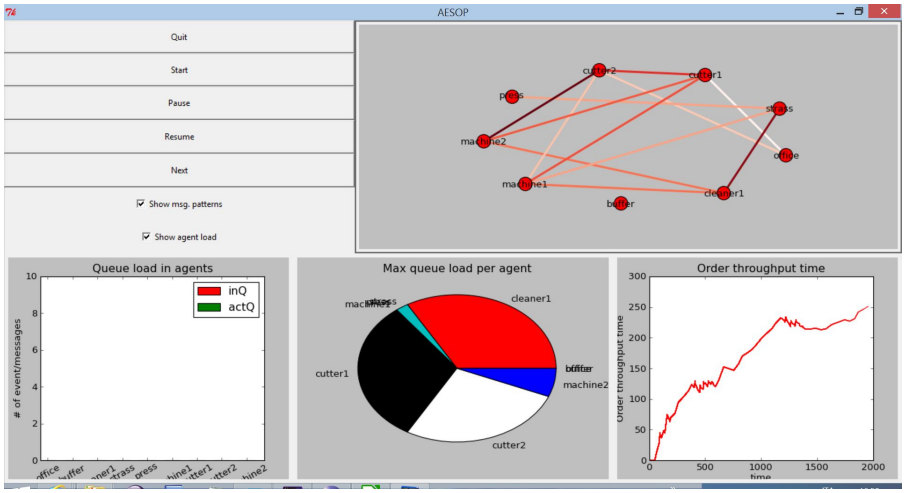


Figure 2: Production GUI in AESOP-ACP framework. The three new widgets at the bottom respectively show: (i) the incoming/in-processing queue load for each agent, (ii) the max load achieved by each agent and (iii) the order throughput time. Nothing is shown in the first new widget since the snapshot has been taken at the end of the simulation.

running of an AESOP-ACP simulation and the standard GUI depicts in real time where is located the highest stress in terms of message exchanges (patterns of exchange) and load over each agent queue (the size of the circle of each agent is proportional to its load).

Unfortunately, the basic GUI does not communicate much information to the user. In fact, it is just a basic class to start to play with.

A better interface is provided by the *aesop.gui.simgui.simGUIProduction*. Now we can try the following:

```
> python startsim.py -d aesop/examples/embroidery/ -s rm_schedule_sec.xls
-G -c aesop.gui.simgui.simGUIProduction
```

By running the simulation, we obtain the results depicted in Figure 2.

The plots highlights the fact that the two cutter agents are the bottlenecks here. Even worst, the throughput time is basically just increasing.

In order to lower the pressure over the cutters we add a new cutter agent, for a total of 3 cutters. We can do that by just uncomment the 'cutter3' definition in the agent graph file. Figure 3 shows the new results for this second experiment.

Unfortunately, the situation in not far from the previous one. The load is split among the cutters, but the throughput is still discouraging.

As a third attempt to find a sustainable set-up, we try an aggressive approach. We add 4 more cutter agents and 2 cleaner agents. Again, it is sufficient to uncomment their definitions in the graph file. Figure 4 shows the third experiment results.

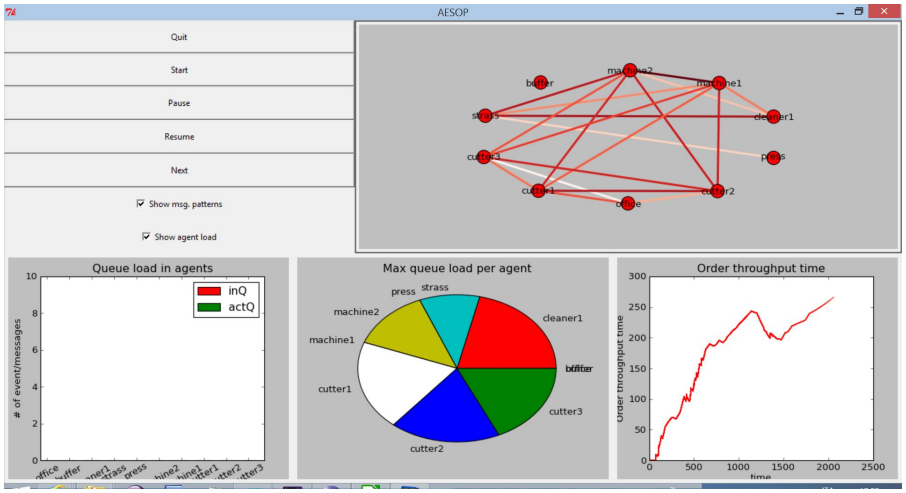


Figure 3: Results achieved by adding a cutter agent to the starting configuration.

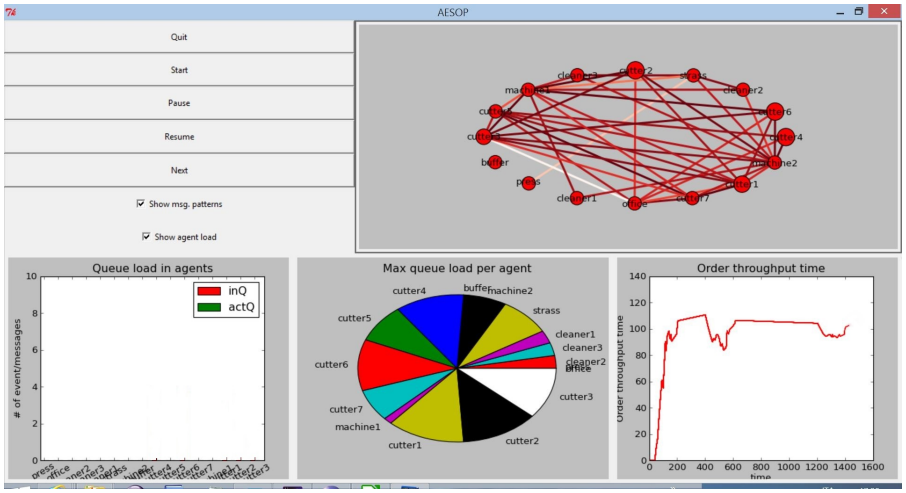


Figure 4: Results achieved by the third aggressive set-up: 7 cutter agents and 3 cleaner agents.

This final configuration achieves that the order throughput time tends to decrease, but oscillates.

This simple example emphasizes how distributed decision making entail a set on non-linearities which are non trivial to be spotted by human beings.