

# How to use MADRAS: a manual for the multiagent distributed resource allocation simulator

October 15, 2007

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Compilation</b>	<b>3</b>
<b>3</b>	<b>The GUI</b>	<b>3</b>
3.1	The Scenario Generator . . . . .	3
3.2	The Experiment Runner . . . . .	5
3.3	The Grapher . . . . .	6
<b>4</b>	<b>Generating and running from the command line</b>	<b>7</b>
4.1	Generating Scenarios . . . . .	7
4.2	Running Experiments . . . . .	8

## 1 Introduction

The MADRAS platform consists of three modules: the Scenario Generator, the Experiment Runner and the Grapher. You can see an overview of the simulation platform in figure 1.

The Scenario Generator is used to describe the society you are modeling: how many agents, how many resources, and most importantly, the utility

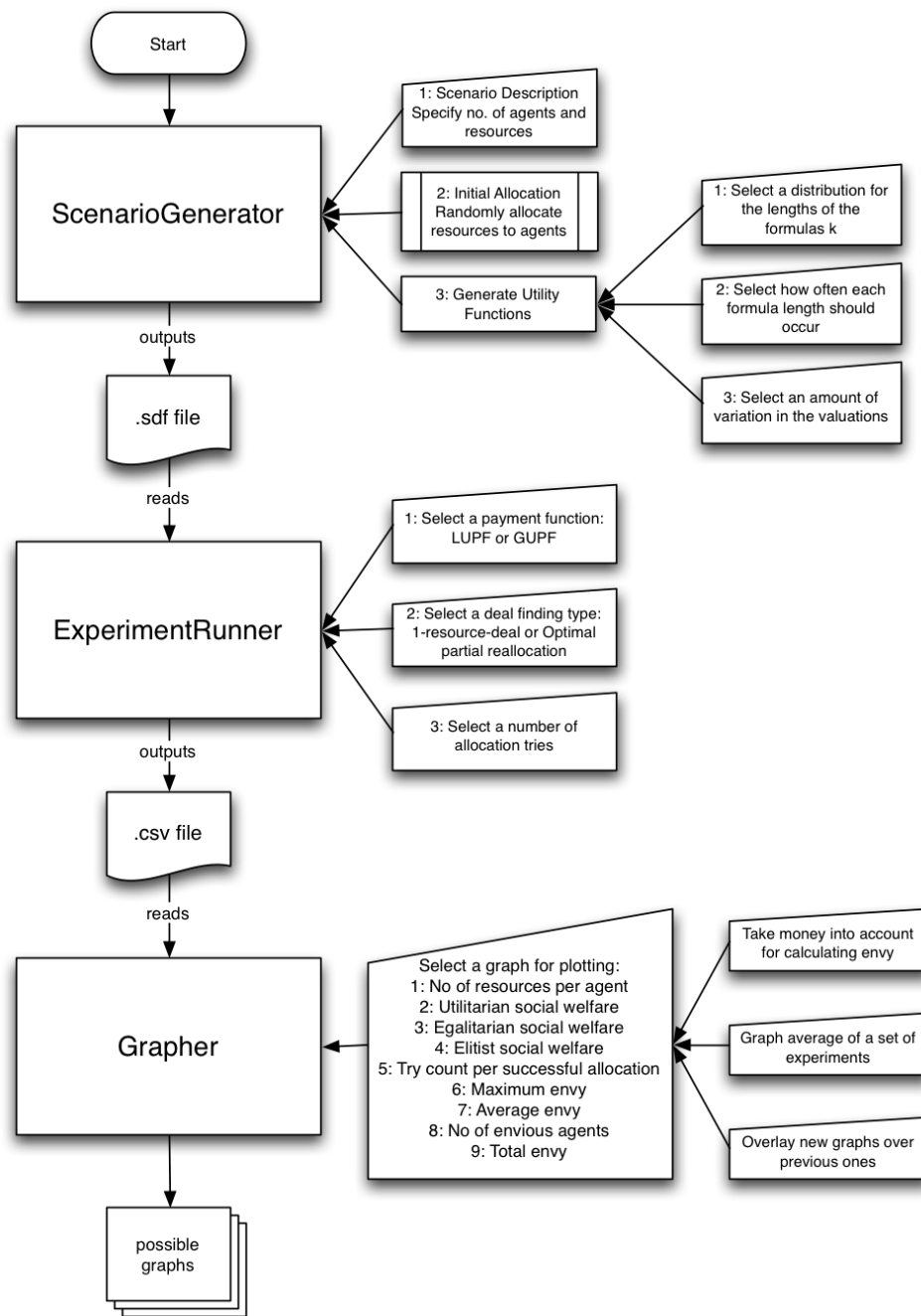


Figure 1: An overview of the various packages in the simulation platform and their input parameters.

functions of those agents (which define how the individual agents feel about the individual resources).

The Experiment Runner then will model negotiation between the agents using either a locally uniform payment function (LUPF) or a globally uniform payment function (GUPF), and either 1-resource deals (where an agent gives exactly 1 resource to another agent) or bilateral deals (where 2 agents exchange as many resources as are desired). All new allocations of resources will be saved in a `.csv` file, where they will be numbered according to the successful attempts.

The Grapher can read the `.csv` file provided by the Experiment Runner and visualize some of the changes taking place between each allocation. This includes plotting the increase or decrease in utilitarian social welfare.

The simulation platform is provided as either source files, or `jars`. Depending on your preference, you can either compile yourself or use the `jar` files.

## 2 Compilation

MADRAS is distributed according to the terms of the GNU General Public License. MADRAS was written in java, using specifically version 1.5.0. If you have java installed, you can compile MADRAS using the source files.

You can compile using the build files supplied. If you use a mac or linux, type `./build.sh`. If you use windows, use `build.bat`.

If after you use the source files you want to combine the source into jars again, you can use `deploy.sh` or `deploy.bat`.

If you would like to use jars instead, simply unzip the `madras_bin.zip` file, and everything should work with double-clicking.

## 3 The GUI

### 3.1 The Scenario Generator

In the scenario generator (figure 2), you define the number of agents, the number of resources and the utility functions. The program might not work if you go over 30 agents and 200 resources.

The generator gives you the option to either generate a new random allocation of resources to agents, or use an existing scenario description file

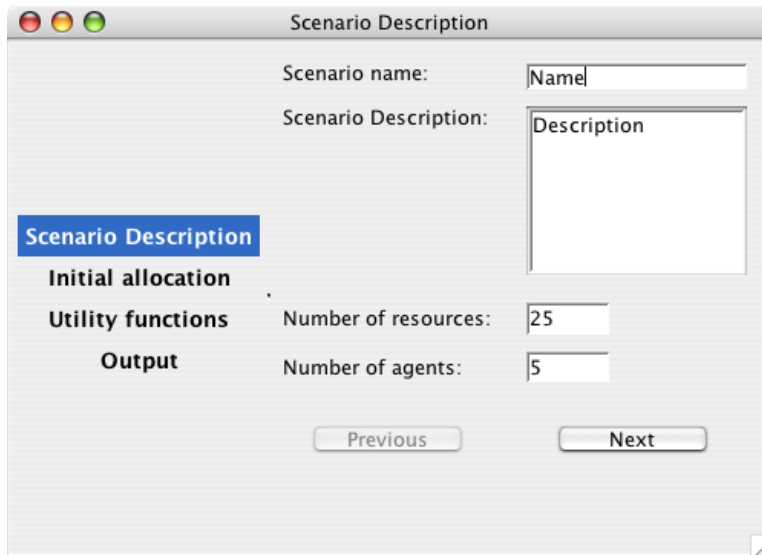


Figure 2: Scenario Description Interface

(.sdf).

To call the `scenario_generator`, double click the `ScenarioGenerator.jar` package or type `java -jar ScenarioGenerator.jar`.

After generating the starting scenario, you can generate utility functions (figure 3.1). The first option is used to specify a distribution for  $k$ , where  $k$  is the maximum size of the set of resources referenced by one goal in an agent's utility function. A precise distribution of  $k = 3$  will result in all agents having utility functions containing between 1 and 3 resources in each goal. An example is  $((\text{and } r_a \text{ } r_b), 5)$ , where the set of resources  $\{r_a, r_b\}$  receives a value of 5. One can also select a normal distribution for  $k$ . Thus, in general this setting defines a distribution, such that every agent is assigned a value  $k$  which is drawn from this distribution. In the preview graph you can see what the distribution will look like using your chosen settings.

The second setting defines a *mapping function* as opposed to a *distribution*. With this parameter the so-called *formula length to count* mapping is specified. This function determines how many formulas will be generated of a certain length. If the selected function returns values that are larger than the maximum amount of possible goals, the maximum is returned instead. When a value below zero is returned, zero is returned. One can choose a lin-

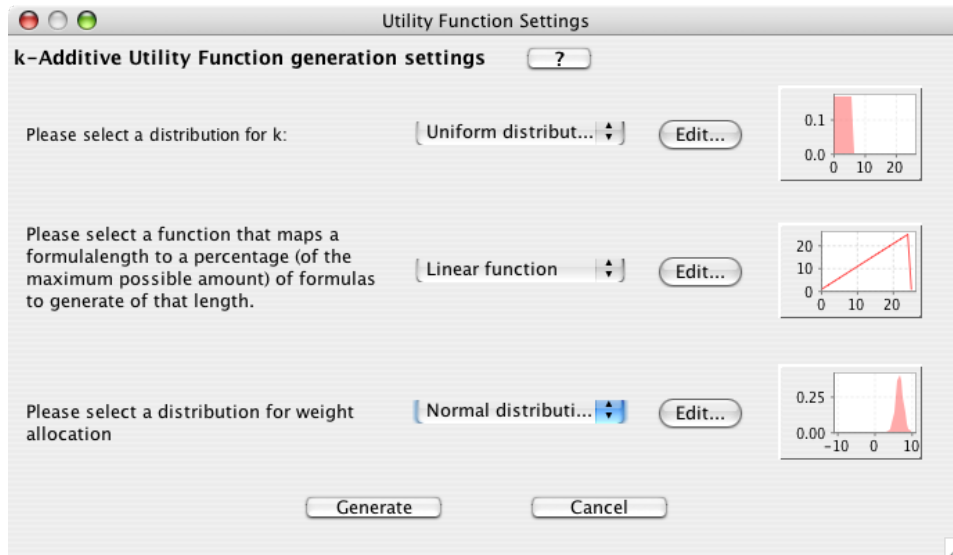


Figure 3: Utility Function Generation Interface

ear percentage function, which returns a percentage of the possible amount of formulas of that length. The linear function does not specify the percentage but the exact amount of formulas. Finally a Gaussian percentage function can be used as well. It should be noted that the total possible amount of formulas is usually very large, and therefore percentage functions will mostly return very high numbers. Of course this is not a problem if a small  $k$  is chosen.

Finally you will need to select the amount of variation in the weights of each formula. Should each set of resources be just as desirable? A common option here is a uniform distribution with both a lower and an upper bound. Do note that you cannot use negative weights, so your lower bound has to be at least 0.

### 3.2 The Experiment Runner

After generating an adequate scenario, you can start to run experiments. In the experiment runner (figure 3.2), you can load your scenario, select the desired payment function and number of allocation tries, and finally run your experiment. You should choose a reasonable amount of allocation tries, for small experiments with less than 20 agents, the 10 000 default is a bit of

an overkill. While running our own experiments, we used a rule of thumb that allocation tries was more or less the total amount of agents times the total amount of resources.

To call the experiment runner, double click on the `ExperimentRunning.jar` package.

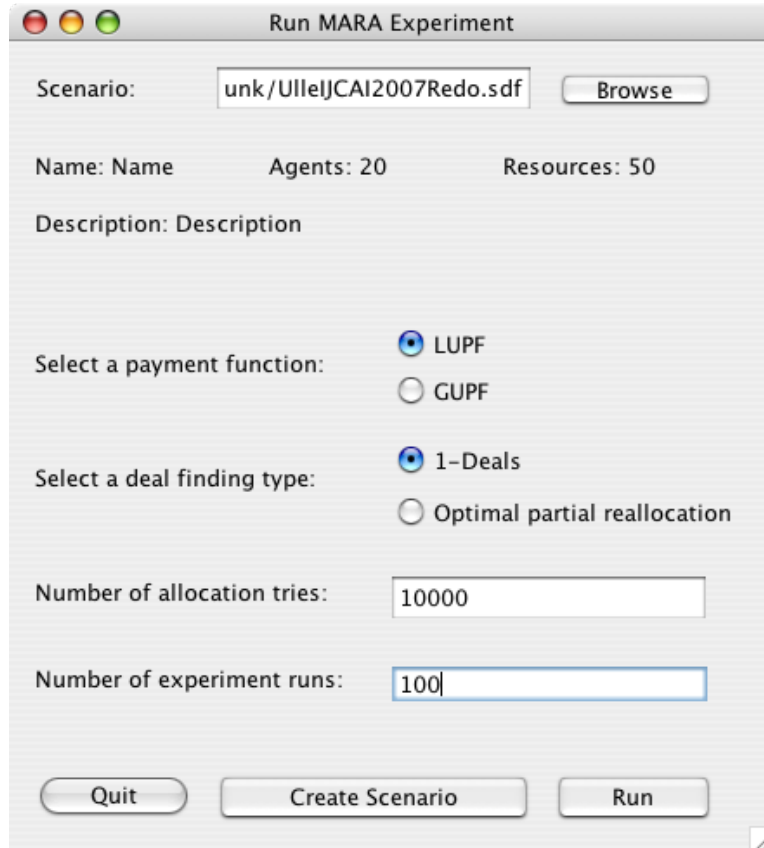


Figure 4: Experiment Running Interface

### 3.3 The Grapher

The experiment running will have generated a `.csv` file. You can read the output yourself, or you can have the Grapher (figure 3.3) plot it into a graph for you.

To run the Grapher, double click on the `Grapher.jar` package or type

`java -jar Grapher.jar` in a command line.

In the Grapher, you can plot the increase in different kinds of social welfare (utilitarian, egalitarian, elitist), how many resources each agent holds at a certain time, how envious your society is at each allocation (with or without money) or how many tries you needed to find a successful allocation. You can either graph each experiment separately, or you can load a bunch of experiments to graph the averages.

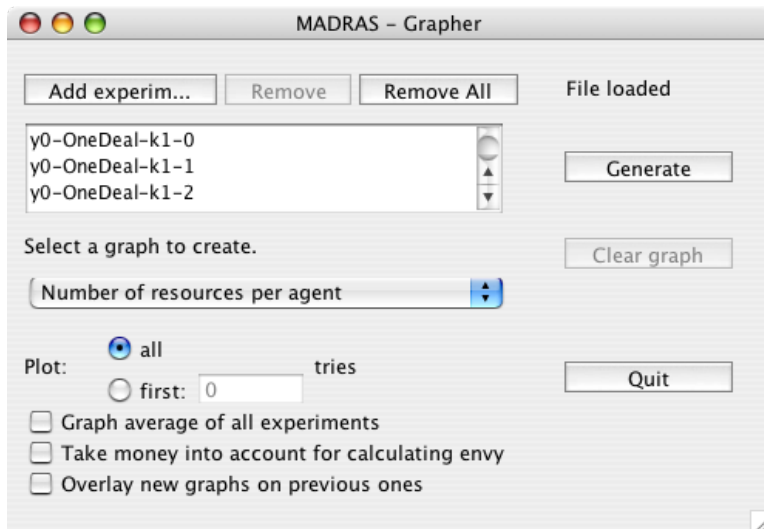


Figure 5: Graph Generation Interface

## 4 Generating and running from the command line

Generating scenarios and running experiments can become a tedious task if you're only varying one or two parameters or trying to run many experiments. You can therefore also create shell scripts to generate scenarios and run experiments for you. To be able to run these though, you will have to compile your own binaries, specific to your operating system.

### 4.1 Generating Scenarios

Scenarios can be generated with the arguments as specified below.

```
java mara.scenario_description.ScenarioCLI --name ExperimentName
```

```
--agents 2 --resources 18 --kdist precise --kdistargs 2 --wdist uniform
--wdistargs 1:100 --fltcm linear --fltcmargs 12:0 --file MyFile.sdf
```

The arguments are pretty straightforward:

`--name` assigns a name to the experiment you are creating.

`--agents` creates the specified number of agents.

`--resources` creates the specified number of resources.

`--kdist` takes a particular kind of distribution for  $k$ . Either `precise`, `uniform` or `normal`.

`--kdistargs` takes the value(s) for your distribution of  $k$ . In the case of `precise`, you specify 1 value. In the case of `uniform`, you specify a lowerbound and upperbound, separated by a colon (e.g. 0:100). In the case of `normal`, you specify a  $\mu$  and  $\sigma$ , again separated by a colon.

`--wdist` takes values for the distribution of weights. The possibilities are also `precise`, `uniform`, or `normal`.

`--wdistargs` adds values for the functions selected in `--wdist`. These are the same as for `--kdist`.

`--fltcm` maps a formula length to a percentage of the maximum amount of formulas you can possibly create or to a specific amount of formulas to generate of that length. Possibilities are `linear`, `linearperc` (Linear percentage function) or `gaussian` (Gaussian percentage function).

`--fltcmargs` In the case of `linear`, you need to specify a specific amount of formulas to create ( $y_0$ ), and a deriv (generally 0), separated by a colon. The same args are used for `linearperc`. In the case of `gaussian`, you specify a  $\mu$  and  $\sigma$ , again separated by a colon.

`--file` names the file you would like to be saving to. The filename should end in `.sdf`.

## 4.2 Running Experiments

Your shell file (for instance a `.sh` file) should contain commands as follows:

```
java mara.experiment_running.ExperimentRunner MyScenario.sdf FileName.csv
PaymentFunction DealFinder AllocationTries NumberOfExperiments
```

The first argument should be a path to the scenario file you would like to run. The second is the name of the file you will be creating. For the

payment function, you can choose either LUPF or GUPF. For the deal finder, you can either use `OneDeal` or `OptimalRealloc`. The `allocations tries` are the number of times MADRAS will attempt to find a new allocation.