EURACE Population GUI - User Guide

Mehmet Gencer

February 13, 2009

This user guide applies to PopGUI software v0.7.6. An up to date copy of this user guide and the latest stable version of software itself can be downloaded from http://ccpforge.cse.rl.ac.uk/projects/eurace/. Latest development snapshots can be obtained from Subversion repository at http://eurace.cs.bilgi.edu.tr/svn/simgui/trunk.

Contents

1 Introduction: Uses and features of PopGUI

PopGUI is written for creating agent populations for EURACE Project[1] simulations using FLAME multi-agent framework. The program reads already designed agent models described in FLAME XMML format, and provides facilities to specify composition of population and specifications for initializing each agent's memory variables. Populations created by the program is then exported as an XML file (so called 0.xml file in FLAME), which is fed into simulation engine which uses these initial conditions and agent behavior described in models to carry out the simulations.

When reading through this guide it is important to distinguish the terms 'population' and 'population instance'. PopGUI's central concern is to maintain specifications for creating initial populations. 'Population' refers to these specifications and the program stores them in its own format in '.pop' files. Many 'population instance's(0.xml files) can be created from the same population specification. Although PopGUI is designed to create valid population instances, the program does not read them back or allow their manipulation.

Population specification involves two main phases. First is population composition. In the FLAME framework an agent population can be divided into any number of sub-populations. For example, the EURACE model contains geographical regions, such as countries, and one can distribute the agents among these regions using the PopGUI. One most straightforward example is countries of the world. Although each country's social life is composed of same types of agents such as workers/consumers, firms, banks, etc., their compositions and demographic features are different since each country have a different population (number of people), number and average size of firms, etc. To account for such realistic populations, PopGUI allows its user to create several regions and specify numbers of each type of agent separately for each region; therefore allowing each region to have a different composition.

The second phase of specifying a population involves assigning values to agents' memory variables. Each type of agent is described by a different set of variables. For example employees can be described with their skill set, income and employment status, whereas a firm is described by number of workers, production capacity, etc. For a realistic population one usually needs standard or empirical random distributions. For instance in our economic world example, sizes of firms in a country usually shows a normal distribution, although its mean is different for each country/region. PopGUI provides a rich set of expressions to specify memory variables.

Another feature of a realistic population is that the agents in it are not isolated but relate to one another. For example workers are employed by firms in their region, or firms borrow money from banks in their country. Therefore these relations must also be initialized in order to create a population, and they must be stored in agent memory variables. In the case of employment relation example, firms will have employees whose skillset is suitable for what

[1]http://www.eurace.org

firm produces, and furthermore the relation is exclusive (i.e. if a person is employed in a firm, he/she cannot be employed by another). In the firm-bank relation example firms usually borrow money from banks in their own region, and the relation is not exclusive (i.e. other firms also borrow money from the same bank). PopGUI provides a rich set of features for creating relations between agents, both exclusive and otherwise.

In additional to those substantial features mentioned above, some rather practical features were also implemented to address needs of agent modelers. For example agent models change through time. New agent types may be added, or agent memory variables are changed. It would be quite impractical to start over with the population design when such changes occur. For this reason PopGUI allows importing population specifications from populations that were worked out for older versions of models. With the help of this feature populations can be re-used with only incremental changes. Such features are introduced in the relevant sections below.

2 Software Requirements and Installation

PopGUI is written in Python language for faster development and portability reasons. It uses GTK+ for its graphical user interface, Therefore PopGUI will run on any platform for which Python (version 2.5 or higher) and GTK+(version 2.0 or higher) is available, including GNU/linux, Microsoft Windows, most Unixes, and others.

The program itself consists of two files only: poplib.py which contains the library functions, and popgui.py which provides the GUI. Follow the instructions below for your operating system to start using PopGUI.

2.1 Installation on GNU/Linux

Install python-gtk2 package. Then place poplib.py and popgui.py into some directory, and run popgui.py. If you want to do these from a command line, do as follows (example for Debian or Ubuntu Linux):

```
$ sudo aptitude install python-gtk2
$ python popgui.py
```

2.2 Manual Installation on Windows

Install the following in order (See `http://www.pygtk.org/downloads.html`http://www.pygtk.org/downloads.h for detailed instructions and links for setting up Python with GTK support):

1. Python 2.5 or newer
2. GTK+ win32 runtime
3. PyCairo

4. PyGObject
5. PyGTK

After that copy poplib.py and popgui.py into a directory, and run popgui.py.

2.3 Installation on Windows using prepackaged installer

EURACE project provides prepackaged installers for installing a set of software tools, including PopGUI. Please check CCPForge website for getting a copy of prepackaged installers: http://ccpforge.cse.rl.ac.uk/

3 Using PopGUI

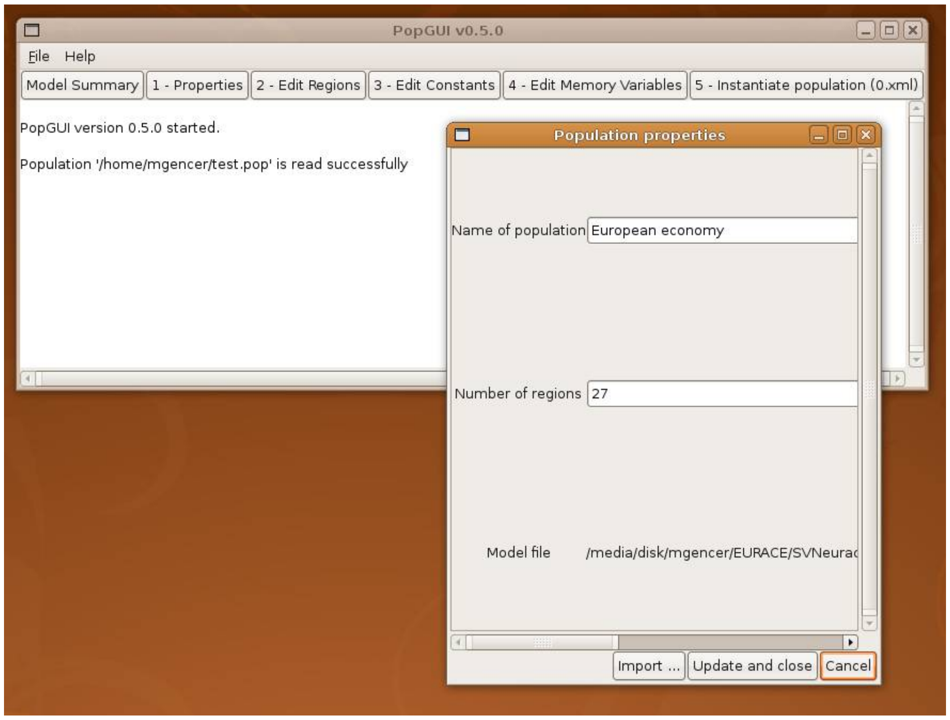
In order to create populations with PopGUI, you need a model describing your agents (their memory and behavior). Since PopGUI was written as a part of EURACE project (see http://www.eurace.org), it currently can use only models given in XMML format, which is the format of the FLAME framework, the official simulation environment of EURACE. Similarly the population instances created by the program are exported only in the XML format adhering to FLAME specifications.

Although agent models contain behavior in addition to memory of agents, only the agent memory is of interest when creating populations. Agent behavior is used later by the simulation engine, along with the population created by PopGUI.

3.1 STEP 1: Creating a population

First create a new population, from file menu. When creating a population you'll be asked for the model xml file which must be in FLAME XMML format. Then you can give a name to the population and set the number of regions in the window that pops up, as shown in Figure 1. The default number o regions is 1. You can increase this number depending on your needs. When you are finished you must save your changes using "Update and close" button in the properties dialog. You can later view or change these properties using the "Properties" button in the toolbar. The update operation only updates the population in your session, and does not save the changes in the file system. To save your changes permanently at any point, you must use "Save population" or "Save population as" menu items from the "File" menu group. The population information is saved in a file with '.pop' extension in your file system. If you want to use a previously created population, click the "Open population" menu in the "File" menu group, then choose a population. Alternatively the name of a '.pop' file can be given as a command line argument to PopGUI when invoking the program (see section 4 about command line options below).

Once you have a population loaded to PopGUI, pressing "Model summary" button will display a rudimentary structure of the model you have chosen, for inspection purposes.



PopGUI version 0.5.0 started.

Population '/home/mgencer/test.pop' is read successfully

Name of population European economy

Number of regions 27

Model file /media/disk/mgencer/EURACE/SVNeurac

Figure 1: Population properties dialog

Note that there's also an "import..." button on the properties dialog. The import feature is provided for a practical concern in modeling work. Once a population is created, PopGUI forgets about the original model and only remembers agents and their memory variables. However the agent models may change while work on population continues. It would be quite impractical to start over entering population specifications every time the model changes, and when the changes are minor (e.g. a few memory variables are added or updated in the model). In such cases what you need to do is to create a new population using the newest model, and then import your previous work into the new population using this "import..." button on the properties menu. However special care must be paid since import operation will import agent memory variable specifications for only those that exists in your latest model. Therefore if there are newly added memory variables or agents, it is up to you to complete the specifications for such new variables and agents.

A restriction of import operation is that number of regions in the current population must match that of the population you are importing from. Therefore even if you plan to change number of regions, you must do so only after importing.

NOTE: As new features are added to PoPGUI it may not stay backwards compatible. For ensuring stability data structures used by PopGUI are given a version number. If you try to open an old population with newer versions of the PopGUI there is a chance that it will fail with a warning about the versions. The recommended way to handle such situations is to create a population from scratch and import memory variables as described above.

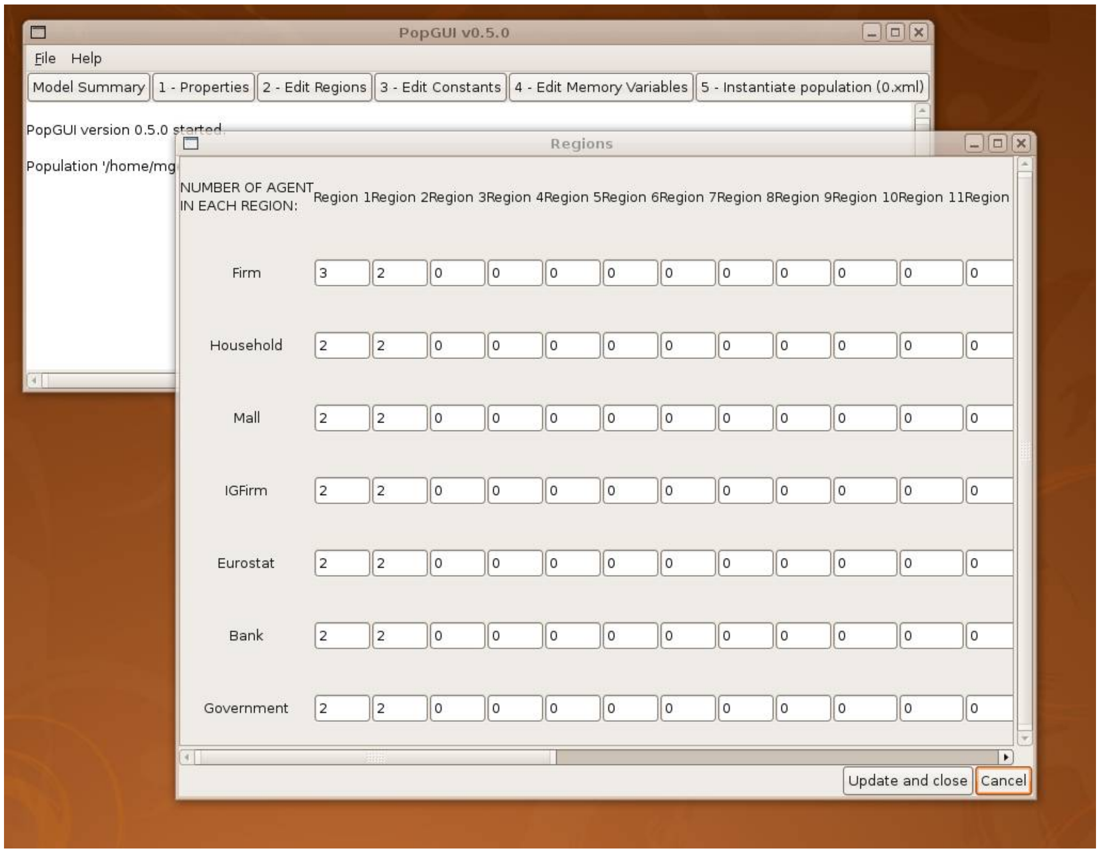
3.2 STEP 2: Specifying population composition

Next step is to set number of each type of agent in each region. When you click "Edit Regions" button in the toolbar, you will be presented a table whose columns are regions and whose rows are agents (see Figure 2). You can enter the number of agents in this grid and then press "update and close" button if you want to save these changes. Your entries must be either positive integers or zero.

3.3 STEP 3: Specifying agent memory variables

This is the most elaborate step of specifying populations in PopGUI. When you press "Edit memory variables" toolbar button, you will be presented with a detailed display of agents and their variables with several buttons at the bottom part of screen, as shown in Figure 3. Agents, their memory variables, and subcomponents of these memory variables are presented in the form of a tree, parts of which can be collapsed or expanded by clicking on the branches. Alternatively all branches can be expanded or collapsed using the "Expand all"/"Collapse All" buttons at the bottom of the screen.

This display also have a grid structure since one must provide specification for memory variables separately for each of the regions. However in most cases

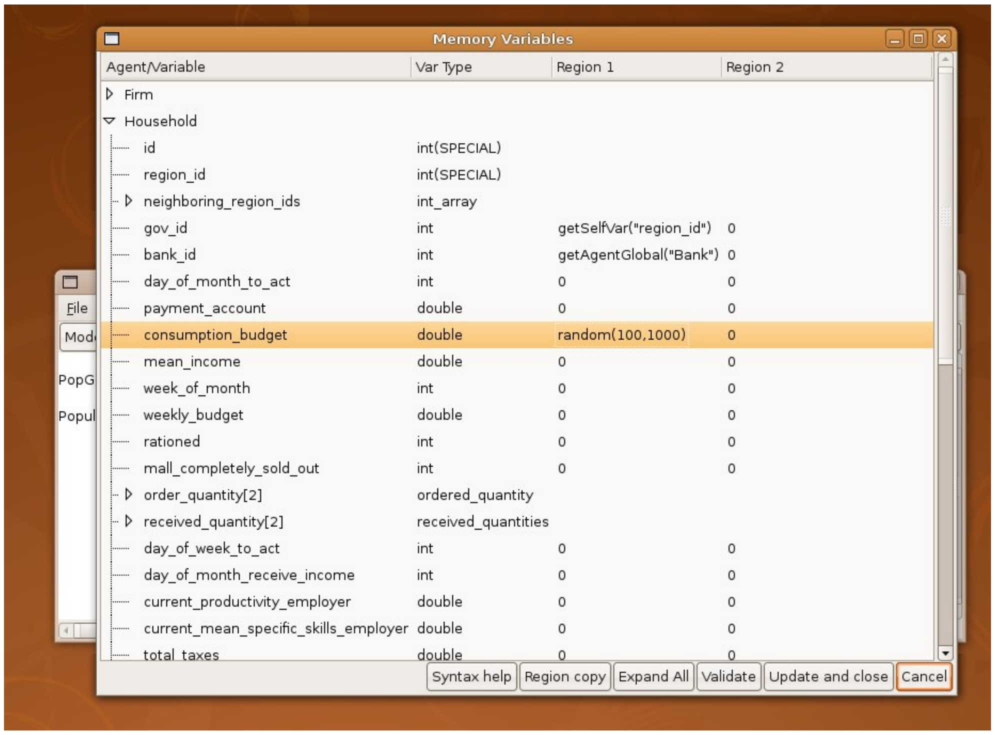


PopGUI version 0.5.0 started.

Population '/home/mg

NUMBER OF AGENT IN EACH REGION:	Region 1	Region 2	Region 3	Region 4	Region 5	Region 6	Region 7	Region 8	Region 9	Region 10	Region 11	Region
Firm	3	2	0	0	0	0	0	0	0	0	0	0
Household	2	2	0	0	0	0	0	0	0	0	0	0
Mall	2	2	0	0	0	0	0	0	0	0	0	0
IGFirm	2	2	0	0	0	0	0	0	0	0	0	0
Eurostat	2	2	0	0	0	0	0	0	0	0	0	0
Bank	2	2	0	0	0	0	0	0	0	0	0	0
Government	2	2	0	0	0	0	0	0	0	0	0	0

Figure 2: Editing population composition



Agent/Variable	Var Type	Region 1	Region 2
Firm			
Household			
id	int(SPECIAL)		
region_id	int(SPECIAL)		
neighboring_region_ids	int_array		
gov_id	int	getSelfVar("region_id")	0
bank_id	int	getAgentGlobal("Bank")	0
day_of_month_to_act	int	0	0
payment_account	double	0	0
consumption_budget	double	random(100,1000)	0
mean_income	double	0	0
week_of_month	int	0	0
weekly_budget	double	0	0
rationed	int	0	0
mall_completely_sold_out	int	0	0
order_quantity[2]	ordered_quantity		
received_quantity[2]	received_quantities		
day_of_week_to_act	int	0	0
day_of_month_receive_income	int	0	0
current_productivity_employer	double	0	0
current_mean_specific_skills_employer	double	0	0
total_taxes	double	0	0

Figure 3: Editing memory variables

the specifications for different regions of the same variable are same or very similar. For this reason a “Region copy” button is provided in this screen. Using this feature you can copy specifications for one region to other regions, and then make incremental changes, to speed up your work.

As described above, specifications of memory variables are elaborate and has two basis: values and relations. Although both are addressed in the language used for memory variable initialization (or ’initform’ as we refer to them in EURACE), different features are required and thus we will present them in separate sections below. The ’syntax help’ button on the memory variables screen displays up-to-date reference on the initform language for your convenience. As a note to users, the initform language is based on dynamic interpretation capability of Python, and hence it has a Pythonic syntax. However. it is not necessary at all for users to have any experience in Python language (although a little bit can help if one wants to use lambda expressions for extending the initform language as will be shown below).

3.3.1 Distinguishing simple and composite variables

Next to each variable’s name in the memory variable editing screen, you will see a variable type. Following are the varieties of these types:

- A few variables are special. Currently these are the ones named ‘id’ and ‘region_id’. Users cannot enter anything for these variables. The ‘id’ is a unique identifier of each agent in the population (for referencing purposes) and it will be assigned sequentially during population instantiation. This variable must exist for FLAME simulations to work. ‘region_id’, when used, will contain an integer which is the number of the region in which the agent is placed.

- Some variables are STATIC or CONSTANT. These appear in cases where their value is fixed in the model or taken from a constant. The names of constants are defined in agent model, but their values are entered in PopGUI using “Edit constants” toolbar button.

- Simple variables has a type which is one of ‘int’ or ‘double’.

- Composite variables are actually C structs. The names and structure of these come from agent model which is used when creating a new population. When a variable is of a composite type, its sub-fields can be expanded and will be entered in separate boxes.

- Finally, a variable can be an array of a simple of composite type. For such variables users will enter a specification for the length of array in addition to the variable (which is used for each element of the array). Special features are provided when one wishes to address all elements of an array, such as when they will be selected from a set without replacement.

Most common use of STATIC or CONSTANT variables occur in array sizes, in cases when array size is fixed in the model either hard coding or by tying to a constant.

3.3.2 Specification syntax for variables and basic random distributions

The entries for memory variables are arithmetic expression which produce a single numeric value (except the special case of array initialization). Therefore the expressions entered into the boxes in this screen can be as simple as a constant number, or a simple arithmetic expression: e.g. "2" or "`2*2-(3.14/4)`". Whenever necessary, PopGUI will convert integer values to real numbers, or vice verse.

In most cases the values will not be constants. Use of random or deterministic initializations are very common. Following functions are provided for use in such cases in initforms:

rand(int,int) : a random integer from the inclusive range.

rand(real,real) : a random real number from the range.

choice([v1,v2,...)]: Pick a value from the list randomly.

normal(mu, sigma) : Normal distribution. mu is the mean, and sigma is the standard deviation.

discrete((probability,value), (probability,value),...) : Choose value given discrete probabilities. Probabilities must add up to 1.0.

Also you can combine any of the above in arithmetic expressions: e.g. "`2+rand(0,5)`" or "`rand(2,6)+choice([1,3,6])`", etc. Discrete versions of random distributions are automatically chosen by PopGUI. In other words even if you use "`random(1.0,2.0)`" for a variable whose type is integer, it will be automatically treated as "`random(1,2)`" by PopGUI.

The expressions you enter are essentially Python expressions which must produce a single value (with the exception of array initialization, see Section 3.3.5). If you have some knowledge of Python, you can enter any valid expression combining the special functions provided by PopGUI.

While entering specifications, the "Syntax help" button on the bottom of this screen can provide you a quick reference help, as an alternative to this user manual. Once you are finished entering specifications you can use the "Validate" button on this screen to ensure that your expressions are correct. This validation is mostly a syntactic one and does not guarantee that you will be able to generate a population instance at the end. That depends on various things including setting up of relations, and its success cannot be determined before you actually proceed to create an instance.

3.3.3 Deterministic initialization

"`deterministic(min,max, lambda function)`" will assign values by taking one value at a time from inclusive range and applying function to selected value. Min and Max must be integer, and Max must be greater than Min. e.g. "`deterministic(0,10,lambda x:x*10)`" will initialize the memory variable of a sequence of agents as:

```
agent[0]=0
agent[1]=10
...
agent[10]=100
agent[11]=0
...
```

3.3.4 Using other memory variables of agent in expressions

It is possible to use value of a memory variable of agent itself in specification of another. The "`getSelfVar()`" function is provided for this purpose. For example if agent has two memory variables of simple types named 'numberofchildren' and 'monthlyexpenses', one can depend on the other by entering "`getSelfVar("numberofchildren")*1250+2000`" in the box for 'monthlyexpenses'.

However you must be careful about dependencies of variables. PopGUI will analyse which other memory variables a variable depends on and sets their values in proper order. But if there are cyclic dependencies which cannot be satisfied, you will face an error at some point.

In the example above "`getSelfVar()`" returned simply a numeric value since the named variable was a simple variable. However, if the memory variable referred to is an array or a composite type, rather than a simple variable, one usually needs to further refer to its elements. For example if the returned value is an array "`getSelfVar("somearray")[0]`" will return first (zero indexed) element of it. If the returned value is a composite variable "`getSelfVar("somecomposite")["x"]`" will return its sub-field named 'x', and so on.

The "`getSelfVar()`" cannot be used to refer to the variable itself, for example to access sibling variables in a data structure, etc. For this purpose another function is provided: "`getSibling("sibling var name")`". When used in an array of data structure, this function will retreive the data field of the current array element. In cases where a data structure contains other data structures or elements, getSibling() can instead access higher levels in the data structure hierarchy using 'level' parameter to climb up the hierarchy. For example "`getSibling("uncle X",level=1)`" and "`getSibling("grand uncle Y",level=2)`" will seek entities at the level of parent and grandparent in the data structure hierarchy, respectively.

3.3.5 Initializing arrays

The only case where the result of your expression is allowed not to be a single numeric value is when you want elements of an array specified at once. In this case your expressions must yield a list whose length is the same with the array length. For example if you have an array of length four, you can use something like “`[1,2,3,4]`” to initialize the array elements sequentially from this list. A few functions whose return values are lists are provided for possible use in such cases:

sample([1,2,3,... ,k)] : Return a k length list of unique elements chosen from the sequence. Used for random sampling without replacement.

permutation([x,y,z,...)]: Return the same list with elements randomly reordered.

sequence(start,end,increment), realsequence(start,end,increment) : generate a sequence of integers with given increment within the inclusive interval: [start, start+increment, start+2*increment, ..., END] where END$\leq$end. rsequence is the version which works for real numbers

3.3.6 Using model constants or population size for scalability purposes

You can access model constants or number of agents in the population using the following:

getConstant(constantname) : Returns the value of model constant. e.g. “`getConstant("alpha")`” to retrieve the value of constant named ”alpha”. (These constant values are what you set in the “Edit constants” tab of the program).

getAgentCountGlobal(agentname) : Get global count of agents with given name, i.e. sum of number of agents in all regions. e.g. “`getAgentCount("Firm")`”

getNumRegions() : Returns the number of regions in the population.

getAgentCountRegional(agentname) : Get regional count of agents with given name.

getAgentIDListRegional(agentname, agentname, ...) : Get a list of agent IDs. You may specify name of one or more agents. Please note that agent names are case sensitive. i.e. if you named an agent as “Firm” you must specify the exact name. The function is deterministic, ie. when called several times the list of agent IDs will be in the same order.

getAgentIDListGlobal(agentname, agentname, ...) : Get a list of agent IDs. You may specify name of one or more agents. Please note that agent names are case sensitive. i.e. if you named an agent as “Firm” you must specify the exact name. The function is deterministic, ie. when called several times the list of agent IDs will be in the same order.

Since the last two functions return a list, rather than a value, they are usually intended for array initialization.

3.3.7 Accessing other agents and initializing agent relations

Agents are not isolated. For example employees and firms in an economics simulation are related to one another through employment relations. To address this aspect of populations PopGUI provides a means to select other agents in the population using some criteria, and use their memory variables in specifying another agent's memory.

In order to pick an agent one can use:

getAgentRegional(agentname, conditions=[("varname",condition), (..), ...)]

getAgentGlobal(agentname, conditions=[...)]

The first one selects an agent from the same region with the referring agent, and the second selects one from the whole population. The selectios are random. Both functions can be provided with zero or more conditions on the agent to be selected using the following format:
"`getAgentRegional("Bank", conditions = [ ("givescredit",equals(1)), ("badreputation",equals(0), ...  ]  )` "
Please note that conditions is a list of tuples `(variable name, condition function)`. Variable name must be a variable of the selected agent, and condition is one of the special functions defined. These functions can be one of the following:

equals(what) : checks whether the value of selected variable equals to the value

between(a,b) : checks whether the value of selected variable within the range

contains(x) is the value of selected variable (which must be a list) contains the value

MooreNeighbour(numcolumns,no) : checks whether the variable (an integer) is in the Moore neighbourhood of "no" in a geography where regions are laid out in rows with a width of 'numcomulmns'. For example if there are 9 regions laid out as follows: 1 2 3 4 5 6 7 8 9 then neighbours of region 1 are 2,5,4, whereas neighbours of region 5 are1,2,3,4,6,7,8,9, etc.

your own functions : Other conditions can be defined using so called lambda functions in Python language, such as "`lambda x:  x<100 and x>=5`". For example "`getAgentRegional("Employee", conditions = [ ("skill",lambda skill :  skill<100 and skill>50) ] )` "

subequals(index,what) : checks whether x[index]==what

subsubequals(index1,index2,what) : checks whether x[index1][index2]==what

Once an agent is selected, its variables can be accessed using a function called "`getAgentVar(variablename)`". For example: "`getAgentGlobal("Bank", conditions = [ ("region_id", MooreNeighbour(10,getSelfVar("region_id") ) ] ).getAgentVar("id")`". If the selected variable is a struct or a list, its elements can be accessed using a syntax similar to the examples given for "`getSelfVar()`", e.g. "`getAgent("Bank").getAgentVar("interestra`

The agents are selected randomly by the current implementation of the above functions in PopGUI to prevent procedural bias in selection.

Exclusive selection If one wishes to prevent others from selecting the same agent, it is possible to do so by setting the 'exclusive' flag to 1 in getAgent variants, i.e. "`getAgentRegional("Employee",exclusive=1)`". Once you do this, the selected employee will never be selected again. However one must be careful since it is possible, depending on the composition of the population, that such conditions may not be satisfied once all agents are taken.

Selecting multiple agents at once Two similar functions allow one to select all agents that match the criteria, instead of only one:

getAllAgentsRegional(agentname,conditions=[])

getAllAgentsGlobal(agentname,conditions=[])

The variants accept same arguments as "`getAgentRegional()`" and "`getAgentGlobal()`". However what is returned is a list of agents. These functions has an optional argument which can be used to turn of random shuffling of the returned agent list. For example getAllAgentsRegional("Firm",randomize=0) will always return the same list in the same order.

If one needs to process specific variables of the agents selected, you will need Pythonic expressions as in the following example:

```
 sum([ agent.getAgentVar("somevar")
   for agent in getAllAgentsRegional("Bank",
         conditions=[("givescredit",equals(1)),("badreputation",equals(0))]) ])
```

The above expression is based on a Python construct which computes a list from elements of a given list, e.g. "`[i*i for i in [1,2,3]]`"

3.3.8 A note about managin dependencies

The group of functions getAgentGlobal/Regional() and getAllAgentsGlobalRegional() create a dependency between agents. In some cases this creates cyclic dependencies which cannot be resolved by the PopGUI. If desired you can solve such situations by using one of the two functions. First is signaling a delayed execution of memory variable initialization:

```
   delayedExecution("some expression"): the expression is executed after all agents and thei
```

to evoid syntax errors due to quotes in your own expressions, use long string syntax in Python by putting your expression in triple quotes. e.g.:

```
   delayedExecution(\"""getAgentGlobal("Bank").getAgentVar("id")\""")
```

Another function is available to fine tune dependencies at the level of variables instead of agents:

```
   dependencyFineTune(type,toagent,tomemvar,expression)
e.g.:
   dependencyFineTune("Global","Government","gov_id",
     \"""getAgentGlobal("Government",conditions=
     [("regions",MooreNeighbour(3,getSelfVar("region_id")))]).getAgentVar("gov_id")""")
```

3.3.9 Some useful Python constructs

Python has a simple syntax and some simple constructs can prove useful in expressing your memory variables. We present some constructs here that are observed to be of common use based on feedback from our users. For further information you are advised to consult Python tutorial at its website `http://www.python.org`.

Constructing lists : range(start,end[,step]) constructs a list which includes start but not end. If given the list is incremented with the given step value.

Lambda functions : An anonymous function can be constructed using lambda function syntax. For example a function to return true if a number is even would be 'lambda x: x

List processing : A new list can be produced by processing numbers in a given list. For example to return squares of a range of numbers 'map(lambda x:x*x, range(1, 11))'.

If what you want is to select only items from a list that match a criteria, use filter. For example to select even numbers from a list: 'filter(lambda x: x

3.3.10 FINISHING UP: Validating and saving memory variable specifications

Before saving memory variables you can use the 'Validate' button on the window, which provides elementary syntax checking of expressions you enter. After validation use "Save and close" button on the memory variables screen to save your entries.

The validation operation only validates expression syntax and does not attempt a full initialization and checking of expressions and references for fitness. Thus you may still catch some problems when you attempt to instantiate population later.

3.4 STEP 4: Editing constants

The constants defined in your model will be assigned a value of zero initially. You can change these values using the “Edit Constant” toolbar button. The constants screen also have a “Validate” button to ensure the expressions you have entered are valid, before you save them.

Instead of numbers, you can also use getAgentCount(”agent name”) as a value for constants.

Unless you set sensible values to constants, it is likely that you will get errors like “division by zero” when you validate memory variable specifivations that use these constants.

3.5 STEP 5: Instantiating population and finishing up

Once you finish entering memory variables and update your changes using the ’save and close’ button, you can create an instance of the population (i.e. the 0.xml file in EURACE parlance) using ’Instantiate population’ button. You will be asked for the name of the file to export the population.

Depending on the number of agents in your population this process can take a long time, and the progress will be displayed on the main screen. You can cancel this operation using the “Cancel” button at any time.

Once you are finished working with your population remember to save it before you quit the program. Later you can re-open them from the file menu, and create new population instances for the same population. Different instances will not be the same but will have same statistical distributions adhering to your specifications.

4 Command line options and debugging

The program accepts a few command line options. Use:

```
python popgui.py -h
```

to display help on these options. Option “-v” will display program version, and “-d” will turn on debugging so that program will dump a lot of messages in the terminal it is being run, which can be helpful in reporting bugs.

Also you can give a population file as command line argument for the program to open the population during start up. For example:

```
python popgui.py -d test.pop
```

will open the population saved in “test.pop” and will turn on debugging while you work.

PopGUI uses versioning for the saved populations to identify the data structure changes that are not backwards compatible, and denies to open populations that were created using past versions of the program and are not compatible with the current version. You can disable version checking using “-i” parameter at the command line, however it is strongly recommended that such usage should be avoided.

5 NOTES

1. **Avoiding dependencies whenever possible**: The group of functions getAgentGlobal/Regional getAllAgentsGlobal/Regional create dependency between agents. We have seen modelers using these functions just to retrieve id’s of other agents. However the correct way to do that is to use getAgentIDListGlobal/Regional. This is because all agents are created and given id’s before any of them are populated with memory variables. Therefore retrieving agent id’s does not generate any dependency, and is a faster operation.

2. **Resolving dependencies**: If you end up with a cyclic dependency, the first thing you can try is to use delayedExecution()

3. **Be careful with Lambda functions**: When describing conditions in agent selection, you may resort to using Python Lambda functions. However you must beware that lambda functions are evaluated when they are called, not when they are created. For example if you want to choose agents whose regionid is same with the choosing agent, you’d try a function like “`lambda x:x["regionid"]==getSelfVar("regionid")`”. However the function is executed within the context of agent being evaluated for selection, because of the inherent way how Python lambda functions are evaluated. (This was the reason that functions such as subequals() subsubequals() were created.)