EURACE Population GUI User Guide

Mehmet Gencer

November 27, 2008

This user guide applies to PopGUI software v0.6.1. Latest stable version of the guide and the software can be downloaded from `http://ccpforge.cse.rl.ac.uk/projects/eurace/`http://ccpforge.cse.rl.ac.uk/. Latest development snapshots can be obtained from Subversion repository at `http://eurace.cs.bilgi.edu.tr/svn/simgui/trunk`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. Whereas 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 we 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 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 such random distributions for memory variables, and use of empirical distributions is planned for future releases.

Another feature of a realistic population is that the agents in it are not isolated but relate to one another. For examples 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 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, and many 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 copy 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:

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

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

2.2 Installation on Windows

Install the following in order (See `http://www.pygtk.org/downloads.html`http://www.pygtk.org/downloads.html 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.

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`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 in the XML format adhering to FLAME specifications only.

Although agent models contain behavior in addition to memory of agents, only the 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 on command line options below).

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

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.

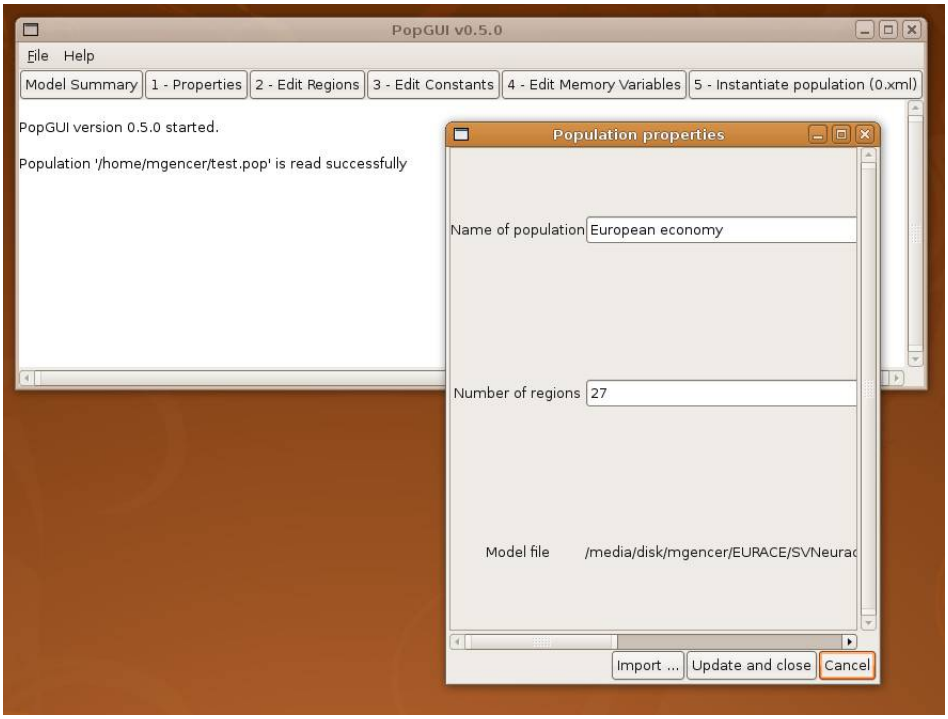

Figure 1: Population properties dialog

NOTE: Since the program is at its development stage it may not be backwards compatible. When you create and save a population, a version number is stored with it indicating version of its data structure. Later if you try to use that population with a 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 sub-components 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 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 user convenience. As a note to users, the initform language is based on dynamic interpretation capability of Python,

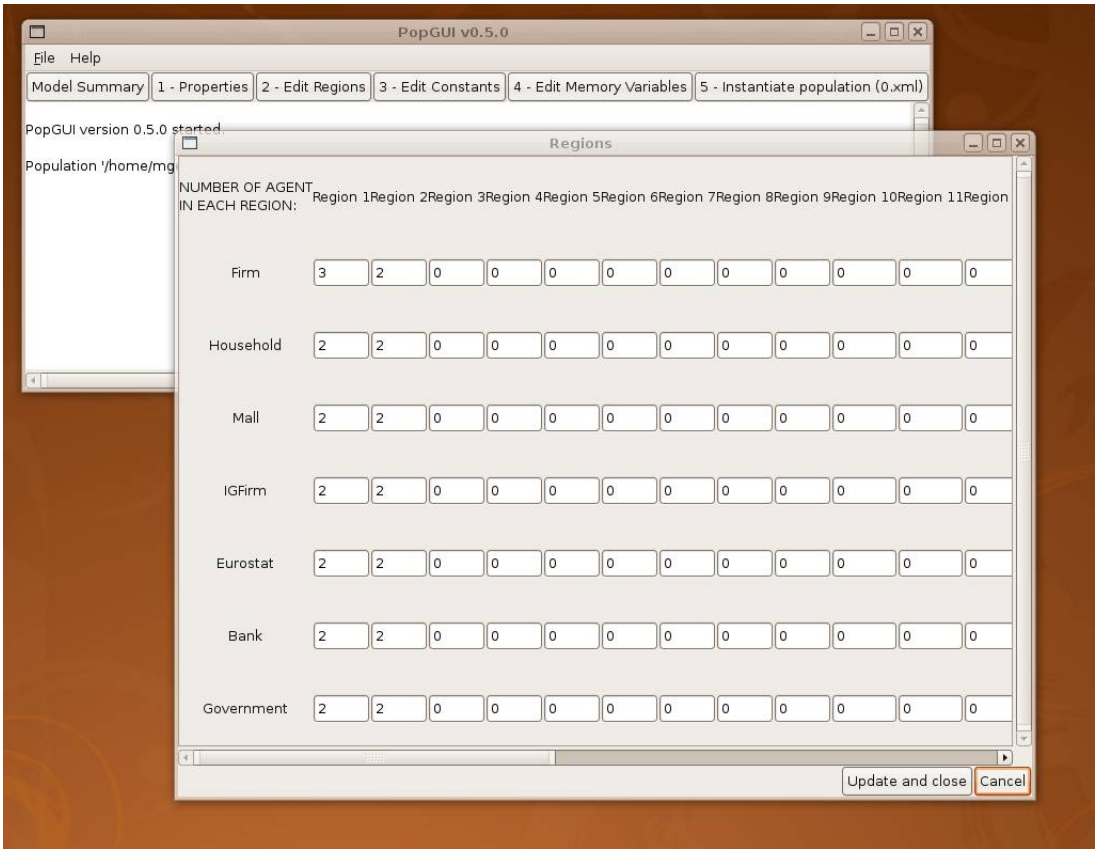

Figure 2: Editing population composition

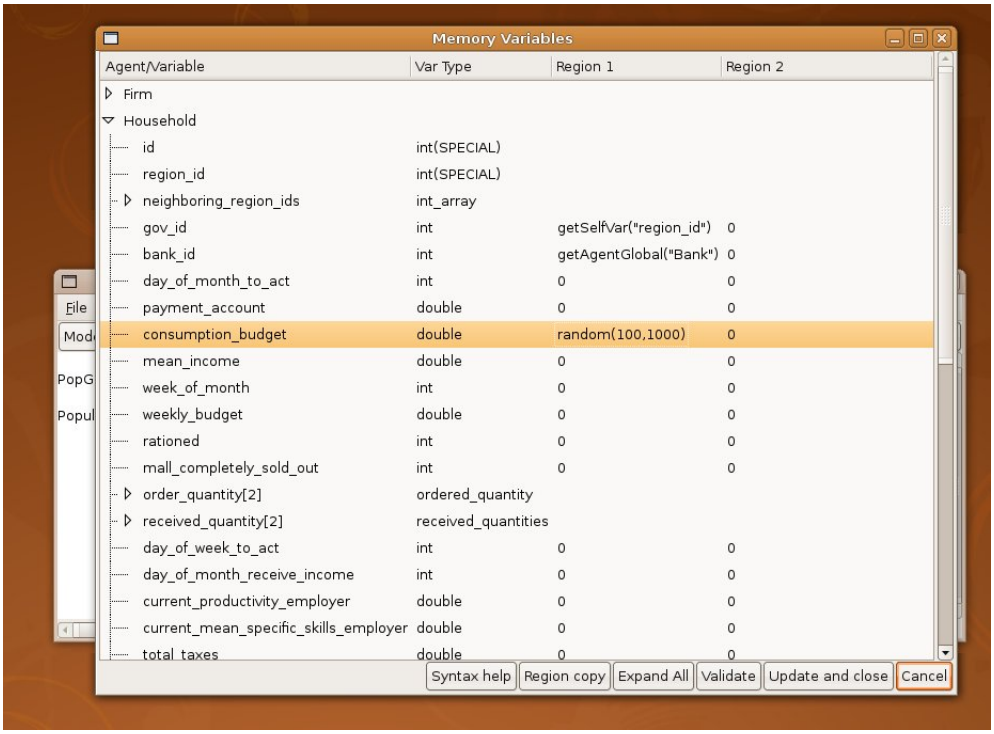

Figure 3: Editing memory variables

and hence it is 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 Specifying values

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.
- 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 of these structs 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, when array size is fixed in the model either hard coding or by tying to a constant.

The entries for memory variables are arithmetic expression which produce a single numeric value (except the special situation 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.

Using random or deterministic distributions In most cases the values are not 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,...)]: Choose 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.

deterministic(min,max, lambda function) : 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
...
```

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.

Using other memory variables 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 using "`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.

Initializing arrays The only case where the result of your expression is not 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. A few functions whose return values are lists are provided for such purposes:

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

sequence(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.

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).

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

3.3.2 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)`" or "`getAgentGlobal(agentname)`" functions, where the former selects an agent from the same region with the referring agent, the latter selects one from the whole population. Both variants can be provided with 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) : is the value of selected variable equals to the value

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

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)])`"

One can use a valid expression as a parameter to these functions. For example "`equals(getSelfVar("id"))`" will work.

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"))]).g`
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("interestrates")["shortterm"]`".

Finally 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.

NOTE: The agents are selected randomly by the current implementation of the above functions in PopGUI.

3.3.3 Validating and saving memory variable specifications

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

3.4 STEP 4: Instantiating population and finishing up

Once you finish entering memory variables update your changes using the 'save and close' button. Finally 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, or modifying other things.

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.