GAMAdocumentation

GAMAteam

Contents

Part I

Platform

Installation and Launching

The GAMA platform can be easily installed in your machine, either if you are using Windows, Mac OS or Ubuntu. GAMA can then be extended by using a number of additional plugins.

This part is dedicated to explain how to install GAMA, launching GAMA and extend the platform by installing additional plugins. All the known issues concerning installation are also explain. The GAMA team provides you a continuous support by proposing corrections to some serious issues through updating patchs. In this part, we will also present you briefly an other way to launch GAMA without any GUI : the headless mode.

Chapter 1

Installation

GAMA 1.7 comes in 5 different versions (32 & 64 bits for Windows & Linux, and 64 bits for MacOS X). You first need to determine which version to use (it depends on your computer, which may, or not, support 64 bits instructions, but also on the version of Java already installed, as the number of bits of the two versions must match).

You can then download the right version from the Downloads page, expand the zip file wherever you want on your machine, and launch GAMA.

Table of contents

System Requirements

GAMA 1.7 requires that Java 1.7 be installed on your machine, approximately 200MB of disk space and a minimum of 4GB of RAM (to increase the portion of memory usable by GAMA, please refer to these instructions).

Installation of Java

On all environments, the recommended Java Virtual Machine under which GAMA has been tested is the one distributed by Oracle (http://www.java.com/en/download/manual.jsp). It may work with others — or not. For better performances, you may also want to install the JDK version of the JVM (and not the standard JRE), although is it not mandatory (GAMA should run fine, but slower, under a JRE).

On MacOS X

The latest version of GAMA requires a JVM (or JDK or JRE) compatible with Java 1.7 to run.

Note for GAMA 1.6.1 users: if you plan to keep a copy of GAMA 1.6.1, you will need to have both Java 1.6 (distributed by Apple) and Java 1.7 (distributed by Oracle) installed at the same time. Because of this bug in SWT (https://bugs.eclipse.org/bugs/show_-bug.cgi?id=374199), GAMA 1.6.1 will not run correctly under Java 1.7 (all the displays will appear empty). To install the JDK 1.6 distributed by Apple, follow the instructions here : http://support.apple.com/kb/DL1572. Alternatively, you might want to go to https://developer.apple.com/downloads and, after a free registration step if you're not an Apple Developer, get the complete JDK from the list of downloads.

On Windows

Please notice that, by default, Internet Explorer and Chrome browsers will download a 32 bits version of the JRE. Running GAMA 32 bits for Windows is ok, but you may want to download the latest JDK instead, in order to both improve the performances of the simulator and be able to run GAMA 64 bits.

- To download the appropriate java version, follow this link: http://www.java.com/en/download/manual.jsp
- Execute the downloaded file
- You can check that a **Java\jre7** (or jre8) folder has been installed at the location **C:\Program Files**

In order for Java to be found by Windows, you may have to modify environment variables:

- Go to the **Control Panel**

- In the new window, go to **System**
- On the left, click on **Advanced System parameters**
- In the bottom, click on **Environment Variables**
- In System Variables, choose to modify the **Path** variable
- At the end, add **;C:\Program Files\Java\jre7\bin** (or jre8\bin)

On Ubuntu & Linux

To have a complete overview of java management on Ubuntu, have a look at:

- Ubuntu Java documentation
- for French speaking users: http://doc.ubuntu-fr.org/java#installations_alternatives

Basically, you need to do:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer (or oracle-java8-
   installer)
```

You can then switch between java version using:

```
sudo update-alternatives --config java
```

See the troubleshooting page for more information on workaround for problems on Ubuntu.

Chapter 2

Launching GAMA

Running GAMA for the first time requires that you launch the application (`Gama.app` on MacOS X, `Gama.exe` on Windows, `Gama` on Linux, located in the folder called `Gama` once you have unzipped the archive). Other folders and files are present here, but you don't have to care about them for the moment. In case you are unable to launch the application, of if error messages appear, please refer to the installation or troubleshooting instructions.

Table of contents

Launching the Application

Note that GAMA can also be launched in two different other ways:

1. In a so-called *headless mode* (i.e. without user interface, from the command line, in order to conduct experiments or to be run remotely). Please refer to the corresponding instructions.

2. From the terminal, using a path to a model file and the name or number of an experiment, in order to allow running this experiment directly (note that the two arguments are optional: if the second is omitted, the file is imported in the workspace if not already present and opened in an editor; if both are omitted, GAMA is launched as usual):

- `Gama.app/Contents/MacOS/Gama path_to_a_model_file#experiment_name_or_number` on MacOS X
- `Gama path_to_a_model_file#experiment_name_or_number` on Linux
- `Gama.exe path_to_a_model_file#experiment_name_or_number` on Windows

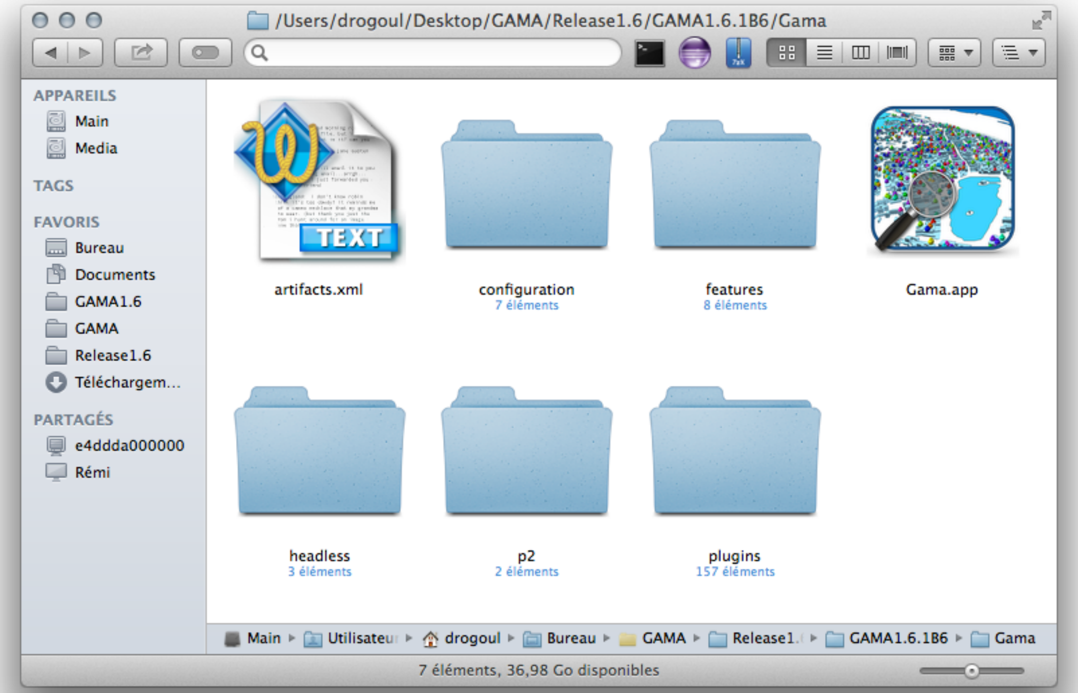
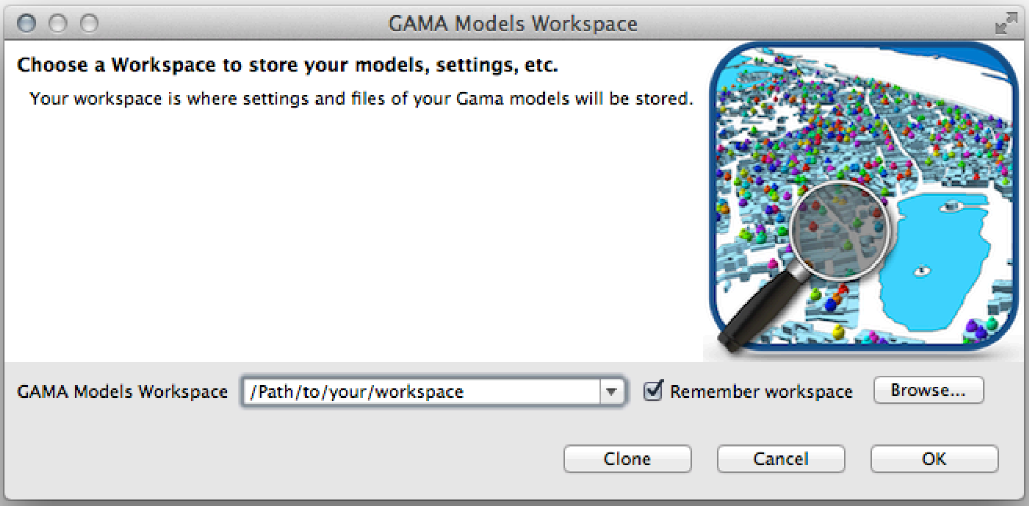


Figure 2.1: Eclipse folder.

Choosing a Workspace

Past the splash screen, GAMA will ask you to choose a workspace in which to store your models and their associated data and settings. The workspace can be any folder in your filesystem on which you have read/write privileges. If you want GAMA to remember your choice next time you run it (it can be handy if you run Gama from the command line), simply check the corresponding option. If this dialog does not show up when launching GAMA, it probably means that you inherit from an older workspace used with GAMA 1.6 or 1.5.1 (and still "remembered"). In that case, a warning will be produced to indicate that the models library is out of date, offering you the possibility to create a new workspace.

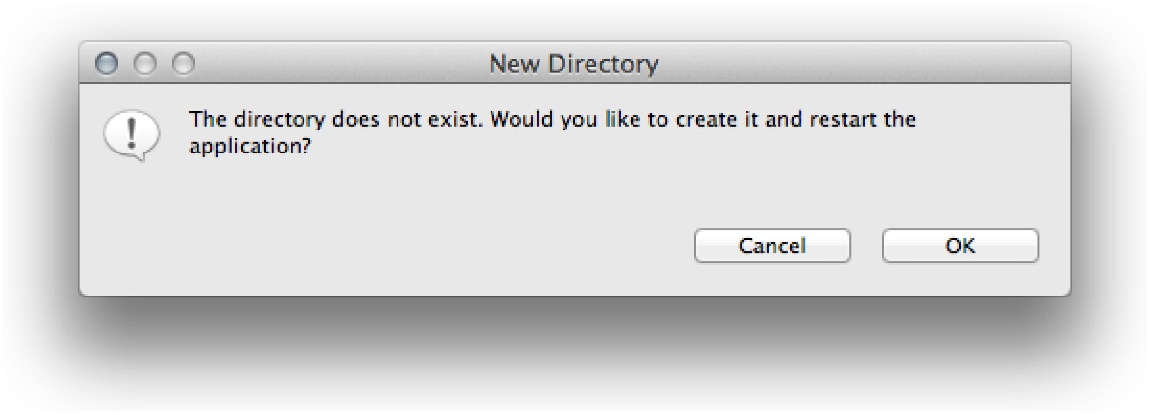


Choose a Workspace to store your models, settings, etc.
Your workspace is where settings and files of your Gama models will be stored.

GAMA Models Workspace /Path/to/your/workspace ☑ Remember workspace

Figure 2.2: Window to choose the workspace.

You can enter its address or browse your filesystem using the appropriate button. If the folder already exists, it will be reused (after a warning if it is not already a workspace). If not, it will be created. It is always a good idea, when you launch a new version of GAMA for the first time, to create a new workspace. You will then, later, be able to import your existing models into it. Failing to do so might lead to odd errors in the various validation processes.



The directory does not exist. Would you like to create it and restart the application?

Figure 2.3: This pop-up appears when the user wants to create a new workspace. Click on OK.

Welcome Page

As soon as the workspace is created, GAMA will open and you will be presented with its **first window**. GAMA is based on Eclipse and reuses most of its visual metaphors for organizing the work of the modeler. The main window is then composed of several **parts**, which can be **views** or **editors**, and are organized in a **perspective**. GAMA proposes 2 main perspectives: *Modeling*, dedicated to the creation of models, and *Simulation*, dedicated to their execution and exploration. Other perspectives are available if you use shared models.

The default perspective in which GAMA opens is *Modeling*. It is composed of a central area where GAML editors are displayed, which is surrounded by a Navigator view on the left-hand side of the window, an Outline view (linked with the open editor) and the Problems view, which indicates errors and warnings present in the models stored in the workspace.

In the absence of previously open models, GAMA will display a *Welcome page* (actually a web page), from which you can find links to the website, current documentation, tutorials, etc. This page can be kept open (for instance if you want to display the documentation when editing models) but it can also be safely closed (and reopened later from the “Views” menu).

From this point, you are now able to edit a new model, navigate in the models libraries, or import an existing model.

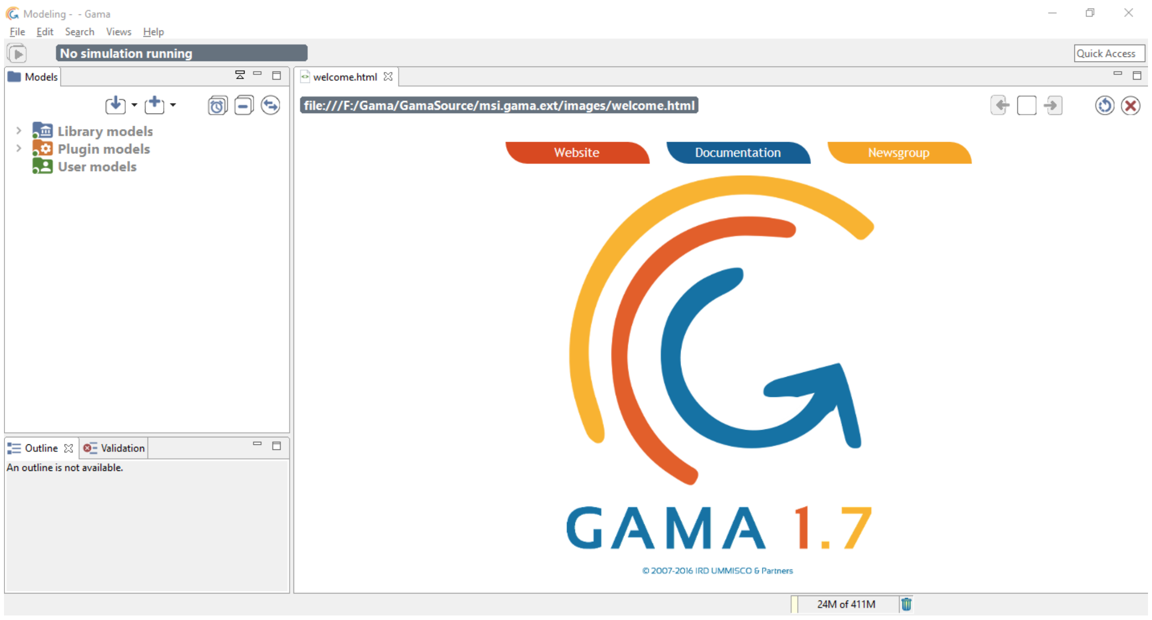


Figure 2.4: GAMA after the first launch.

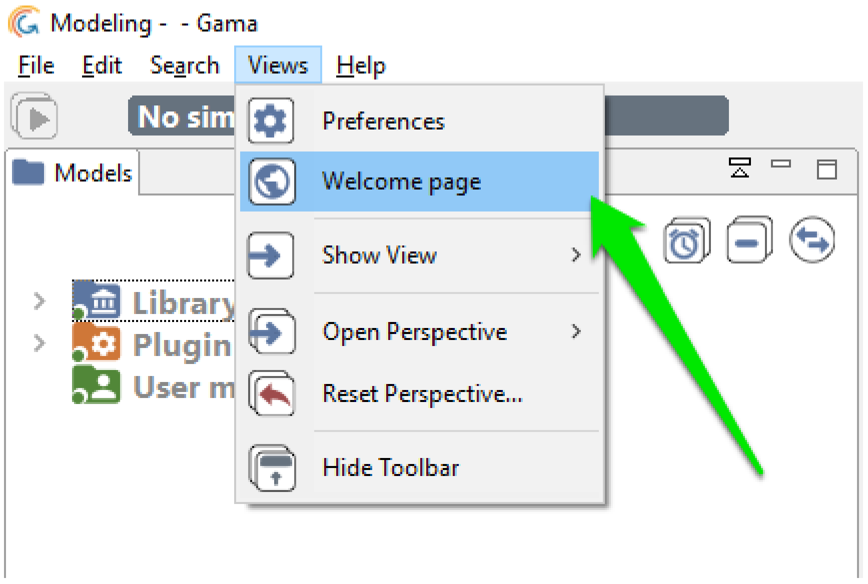


Figure 2.5: Menu to open new views.

Chapter 3

Headless Mode

The aim of this feature is to be able to run one or multiple instances of GAMA without any user interface, so that models and experiments can be launched on a grid or a cluster. Without GUI, the memory footprint, as well as the speed of the simulations, are usually greatly improved.

In this mode, GAMA can only be used to run experiments and that editing or managing models is not possible. In order to launch experiments and still benefit from a user interface (which can be used to prepare headless experiments), launch GAMA normally (see here) and refer to this page for instructions.

Table of contents

Command

There are two ways to run a GAMA experiment in headless mode: using a dedicated shell script (recommended) or directly from the command line. These commands take 2 arguments: an experiment file and an output directory.

Shell Script

It can be found in the `headless` directory located inside `Gama`. Its name is `gama-headless.sh` on MacOSX and Linux, and `gama-headless.bat` on Windows.

```
sh gama-headless.sh [m/c/t/hpc/v] $1 $2
```

- with:
  - $1 input parameter file : an xml file determining experiment parameters and attended outputs
  - $2 output directory path : a directory which contains simulation results (numerical data and simulation snapshot)
  - options [-m/c/t/hpc/v]
    * -m memory : memory allocated to gama
    * -c : console mode, the simulation description could be written with the stdin
    * -t : tunneling mode, simulation description are read from the stdin, simulation results are printed out in stdout
    * -hpc nb_of_cores : allocate a specific number of cores for the experiment plan
    * -v : verbose mode. trace are displayed in the console
- For example (using the provided sample), navigate in your terminal to the GAMA root folder and type :

```
sh headless/gama-headless.sh headless/samples/predatorPrey.xml
   outputHeadLess
```

As specified in **predatorPrey.xml**, this command runs the prey - predator model for 1000 steps and record a screenshot of the main display every 5 steps. The screenshots are recorded in the directory outputHeadLess (under the GAMA root folder).

Not that the current directory to run gama-headless command must be $GAMA_PATH/-headless

Java Command

```
java -cp $GAMA_CLASSPATH -Xms512m -Xmx2048m -Djava.awt.headless=
   true org.eclipse.core.launcher.Main -application msi.gama.
   headless.id4 $1 $2
```

- with:
  - $GAMA_CLASSPATH gama classpath: contains relative or absolute path of jars inside the gama plugin directory and jars created by users
  - $1 input parameter file: an xml file determining experiment parameters and attended outputs
  - $2 output directory path: a directory which contains simulation results (numerical data and simulation snapshot)

Note that the output directory is created during the experiment and should not exist before.

Experiment Input File

The XML input file contains for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Experiment_plan>
 <Simulation id="2" sourcePath="./predatorPrey/predatorPrey.gaml"
    finalStep="1000" experiment="predPrey">
  <Parameters>
```

```
    <Parameter name="nb_predator_init" type="INT" value="53" />
    <Parameter name="nb_preys_init" type="INT" value="621" />
  </Parameters>
  <Outputs>
    <Output id="1" name="main_display" framerate="10" />
    <Output id="2" name="number_of_preys" framerate="1" />
    <Output id="3" name="number_of_predators" framerate="1" />
    <Output id="4" name="duration" framerate="1" />
  </Outputs>
 </Simulation>
</Experiment_plan>
```

Note that several simulations could be determined in one experiment plan. These simulations are run in parallel according to the number of allocated cores.

Heading

```
<Simulation id="2" sourcePath="./predatorPrey/predatorPrey.gaml"
   finalStep="1000" experiment="predPrey">
```

- with:
  - `id`: permits to prefix output files for experiment plan with huge simulations.
  - `sourcePath`: contains the relative or absolute path to read the gaml model.
  - `finalStep`: determines the number of simulation step you want to run.
  - `experiment`: determines which experiment should be run on the model. This experiment should exist, otherwise the headless mode will exit.

Parameters

One line per parameter you want to specify a value to:

```
<Parameter name="nb_predator_init" type="INT" value="53" />
```

- with:
  - `name`: name of the parameter in the gaml model
  - `type`: type of the parameter (INT, FLOAT, BOOLEAN, STRING)
  - `value`: the chosen value

Outputs

One line per output value you want to retrieve. Outputs can be names of monitors or displays defined in the 'output' section of experiments, or the names of attributes defined in the experiment or the model itself (in the 'global' section).

```
 ... with the name of a monitor defined in the 'output'
section of the experiment...
 <Output id="2" name="number_of_preys" framerate="1" />
 ... with the name of a (built-in) variable defined in the
experiment itself...
 <Output id="4" name="duration" framerate="1" />
```

- with:
  - `name` : name of the output in the 'output'/'permanent' section in the experiment or name of the experiment/model attribute to retrieve
  - `framerate` : the frequency of the monitoring (each step, each 2 steps, each 100 steps...).
- Note that :
  - the lower the framerate value the longer the experiment.
  - if the chosen output is a display, an image is produced and the output file contains the path to access this image

Output Directory

During headless experiments, a directory is created with the following structure:

```
Outputed-directory-path/
    |-simulation-output.xml
    |- snapshot
          |- main_display2-0.png
          |- main_display2-10.png
          |- ...
```

- with:

  - `simulation-output.xml`: containing the results
  - `snapshot`: containing the snapshots produced during the simulation

Is it possible to change the output directory for the images by adding the attribute "output_-path" in the xml :

If we write `<Output id="1" name="my_display" file:"/F:/path/imageName" framerate="10" />`, then the display "my_display" will have the name "imageName-stepNb.png" and will be written in the folder "/F:/path/"

Simulation Output

A file named `simulation-output.xml` is created with the following contents when the experiment runs.

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation id="2" >
    <Step id='0' >
        <Variable name='main_display' value='main_display2-0.png
   '/>
        <Variable name='number_of_preys' value='613'/>
        <Variable name='number_of_predators' value='51'/>
                <Variable name='duration' value='6' />
    </Step>
    <Step id='1' >
        <Variable name='main_display' value='main_display2-0.png
   '/>
        <Variable name='number_of_preys' value='624'/>
        <Variable name='number_of_predators' value='51'/>
                <Variable name='duration' value='5' />
    </Step>
        <Step id='2'>

...
```

- With:
  - `<Simulation id="2" >`: block containing results of the simulation 2 (this Id is identified in the Input Experiment File)

  - `<Step id='1' > ... </Step>`: one block per step done. The id corresponds to the step number

Step

```
 <Step id='1' >
     <Variable name='main_display' value='main_display2-0.png
'/>
     <Variable name='number_of_preys' value='624'/>
     <Variable name='number_of_predators' value='51'/>
             <Variable name='duration' value='6' />
 </Step>
```

There is one Variable block per Output identified in the output experiment file.

Variable

```
<Variable name='main_display' value='main_display2-0.png'/>
```

- with:
  - `name`: name of the output, the model variable
  - `value`: the current value of model variable.

Note that the value of an output is repeated according to the framerate defined in the input experiment file.

Snapshot files

This directory contains images generated during the experiment. There is one image per displayed output per step (according to the framerate). File names follow a naming convention, e.g:

```
 [outputName][SimulationID]_[stepID].png -> main_display2-20.
 png
```

Note that images are saved in '.png' format.

Chapter 4

Updating GAMA

Unless you are using the version of GAMA built from the sources available in the GIT repository of the project (see here), you are normally running a specific **release** of GAMA that sports a given **version number** (e.g. GAMA 1.6.1, GAMA 1.7, etc.). When new features were developed, or when serious issues were fixed, the release you had on your disk, prior to GAMA 1.6.1, could not benefit from them. Since this version, however, GAMA has been enhanced to support a *self_update* mechanism, which allows to import from the GAMA update site additional plugins (offering new features) or updated versions of the plugins that constitute the core of GAMA.

Table of contents

Manual Update

To activate this feature, you have to invoke the "Check for Updates" or "Install New Software..." menu commands in the "Help" menu.

The first one will only check if the existing plugins have any updates available, while the second will, in addition, scan the update site to detect any new plugins that might be added to the current installation.

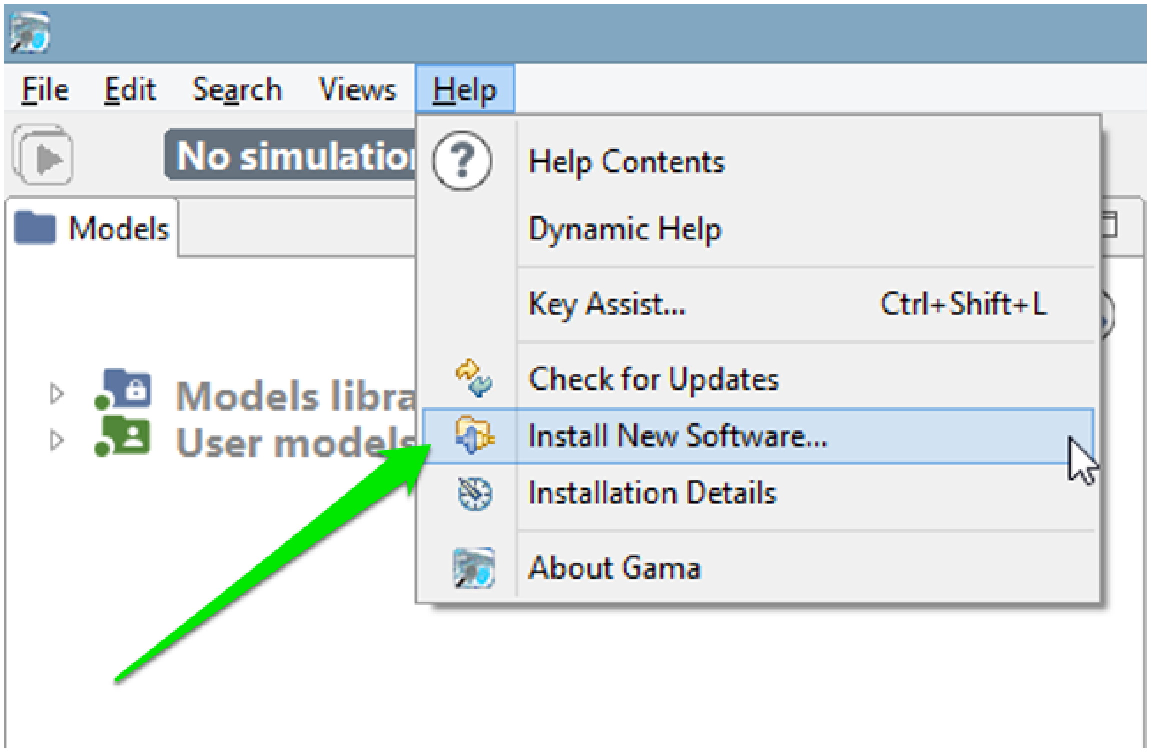


Figure 4.1: Menu to install new extensions to GAMA.

In general, it is preferable to use the second command, as more options (including that of *desinstalling* some plugins) are provided. Once invoked, it makes the following dialog appear:

GAMA expects the user to enter a so-called *update site*. You can copy and paste the following line (or choose it from the drop-down menu as this address is built inside GAMA):

```
http://updates.gama-platform.org
```

GAMA will then scan the entire update site, looking both for new plugins (the example below) and updates to existing plugins. The list available in your installation will of course be different from the one displayed here.

Choose the ones you want to install (or update) and click "Next...". A summary page will appear, indicating which plugins will actually be installed (since some plugins might require

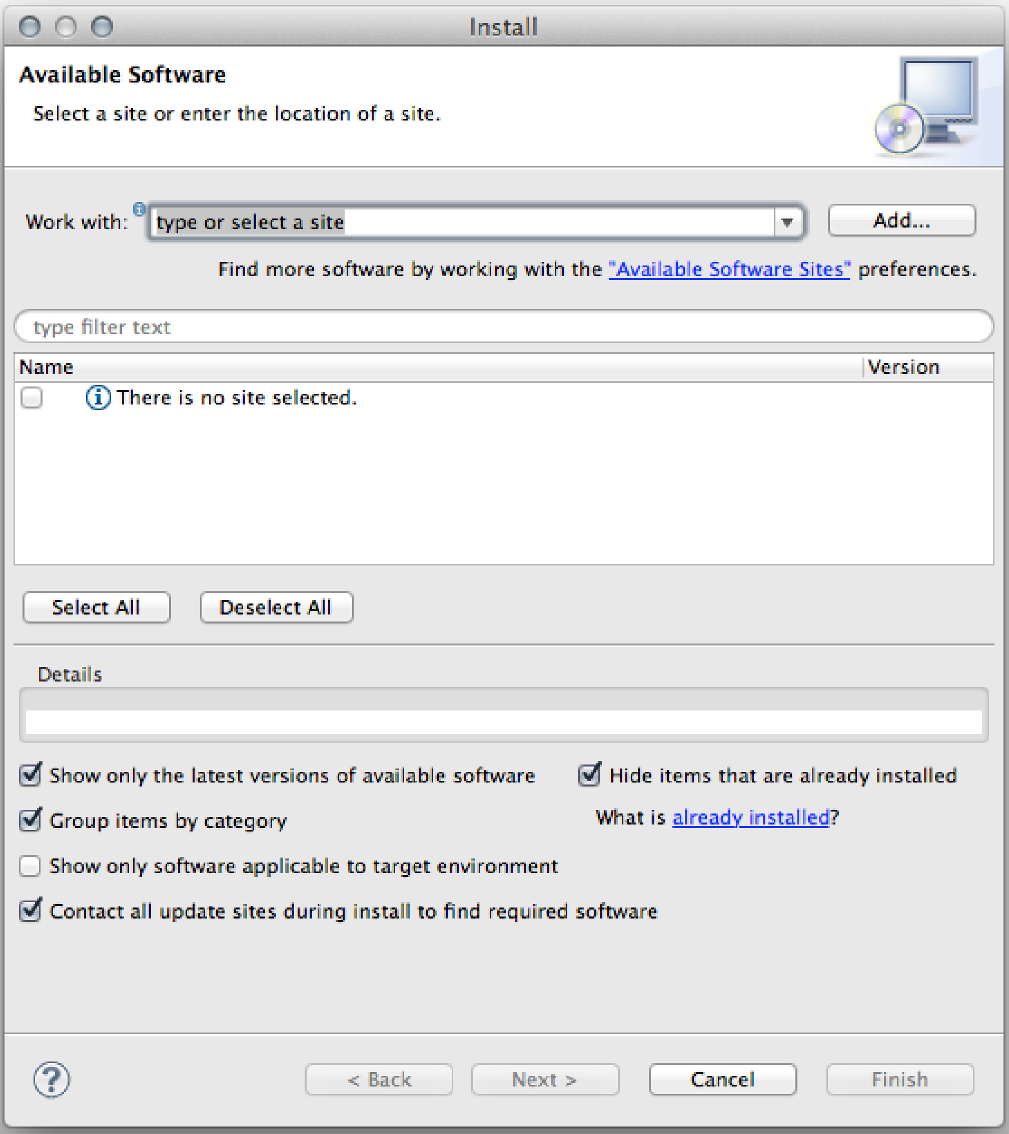
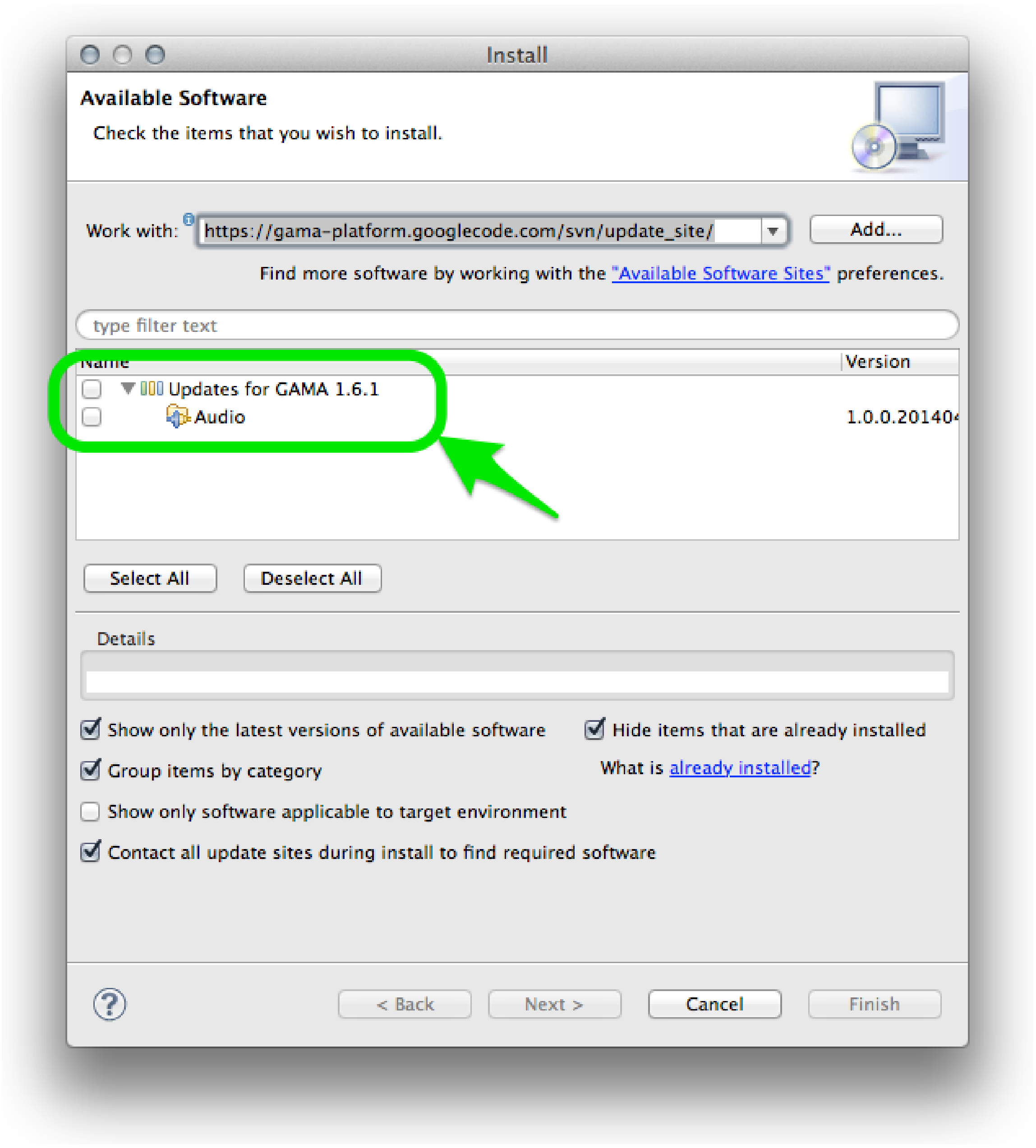

Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish

Figure 4.2: Window where the user enters the adress of an update site and can choose plug-ins to install.


Install

Available Software

Check the items that you wish to install.

Work with: https://gama-platform.googlecode.com/svn/update_site/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ Updates for GAMA 1.6.1	
☐ Audio	1.0.0.20140

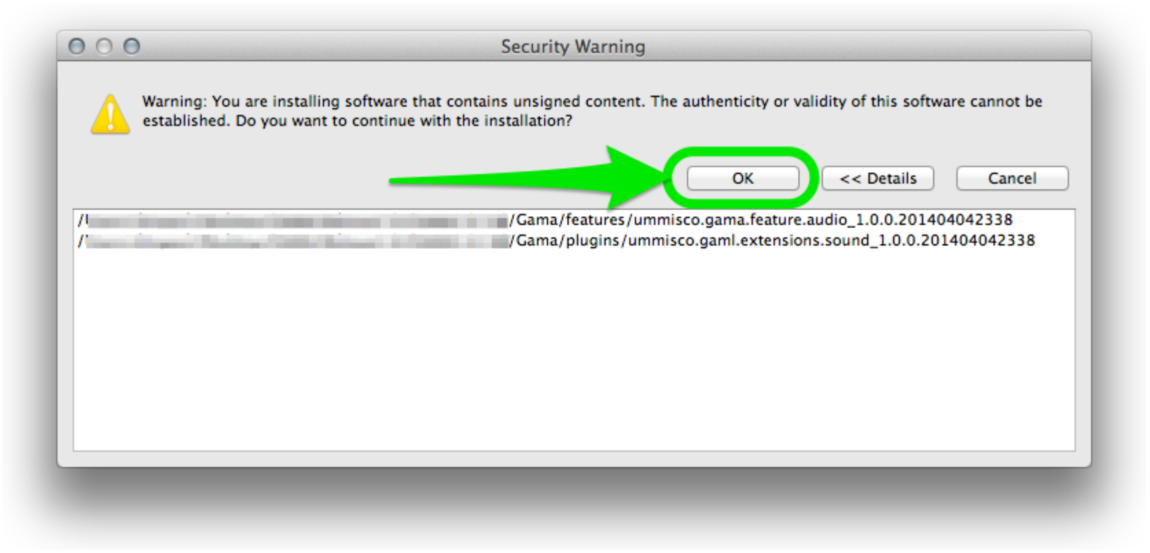
Details

☑ Show only the latest versions of available software
☑ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

Figure 4.3: Display of the list of available extensions.

additional plugins to run properly), followed by a license page that you have to accept. GAMA will then proceed to the installation (that can be cancelled any time) of the plugins chosen.

During the course of the installation, you might receive the following warning, that you can dismiss by clicking "OK".


Security Warning

Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

```
/[redacted]/Gama/features/ummisco.gama.feature.audio_1.0.0.201404042338
/[redacted]/Gama/plugins/ummisco.gaml.extensions.sound_1.0.0.201404042338
```

Figure 4.4: Warning window that can be dismissed.

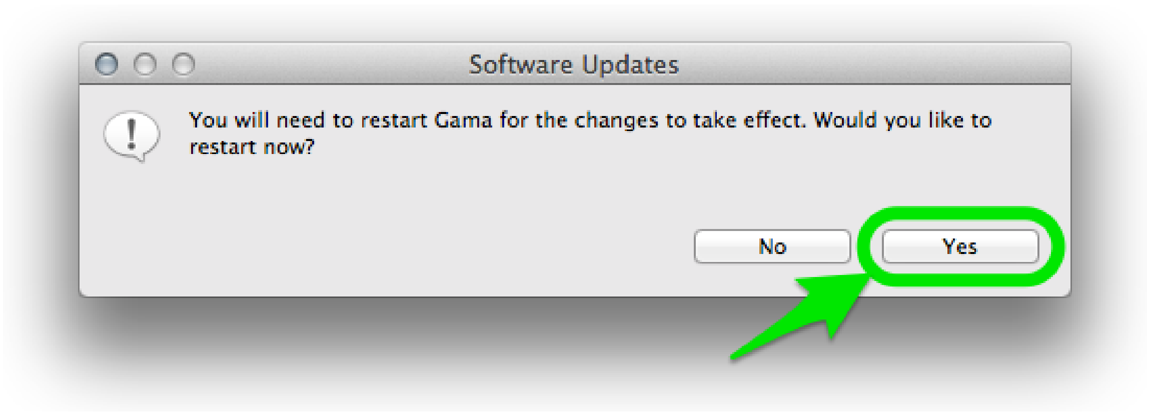
Once the plugins are installed, GAMA will ask you whether you want to restart or not. It is always safer to do so, so select "Yes" and let it close by itself, register the new plugins and restart.

Automatic Update

GAMA offers a mechanism to monitor the availability of updates to the plugins already installed. To install this feature, open the preferences of GAMA and choose the button "Advanced…", which gives access to additional preferences.

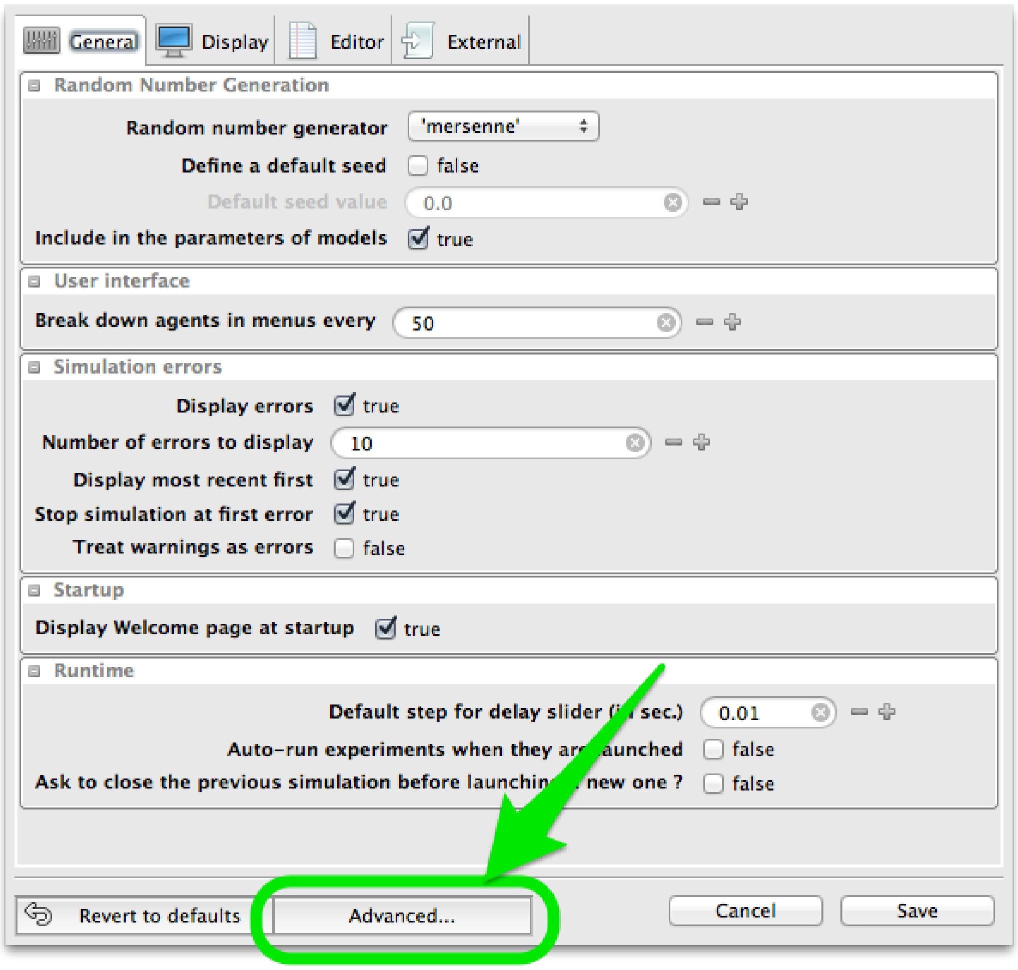
In the dialog that appears, navigate to "Install/Update > Automatic Updates". Then, enable the option using the check-box on the top of the dialog and choose the best settings for your workflow. Clicking on "OK" will save these preferences and dismiss the dialog.

From now on, GAMA will continuously support you in having an up-to-date version of the platform, provided you accept the updates.



You will need to restart Gama for the changes to take effect. Would you like to restart now?

Figure 4.5: After installation, GAMA has to be restarted.



Random Number Generation

Random number generator: 'mersenne'
Define a default seed: false
Default seed value: 0.0
Include in the parameters of models: true

User interface

Break down agents in menus every: 50

Simulation errors

Display errors: true
Number of errors to display: 10
Display most recent first: true
Stop simulation at first error: true
Treat warnings as errors: false

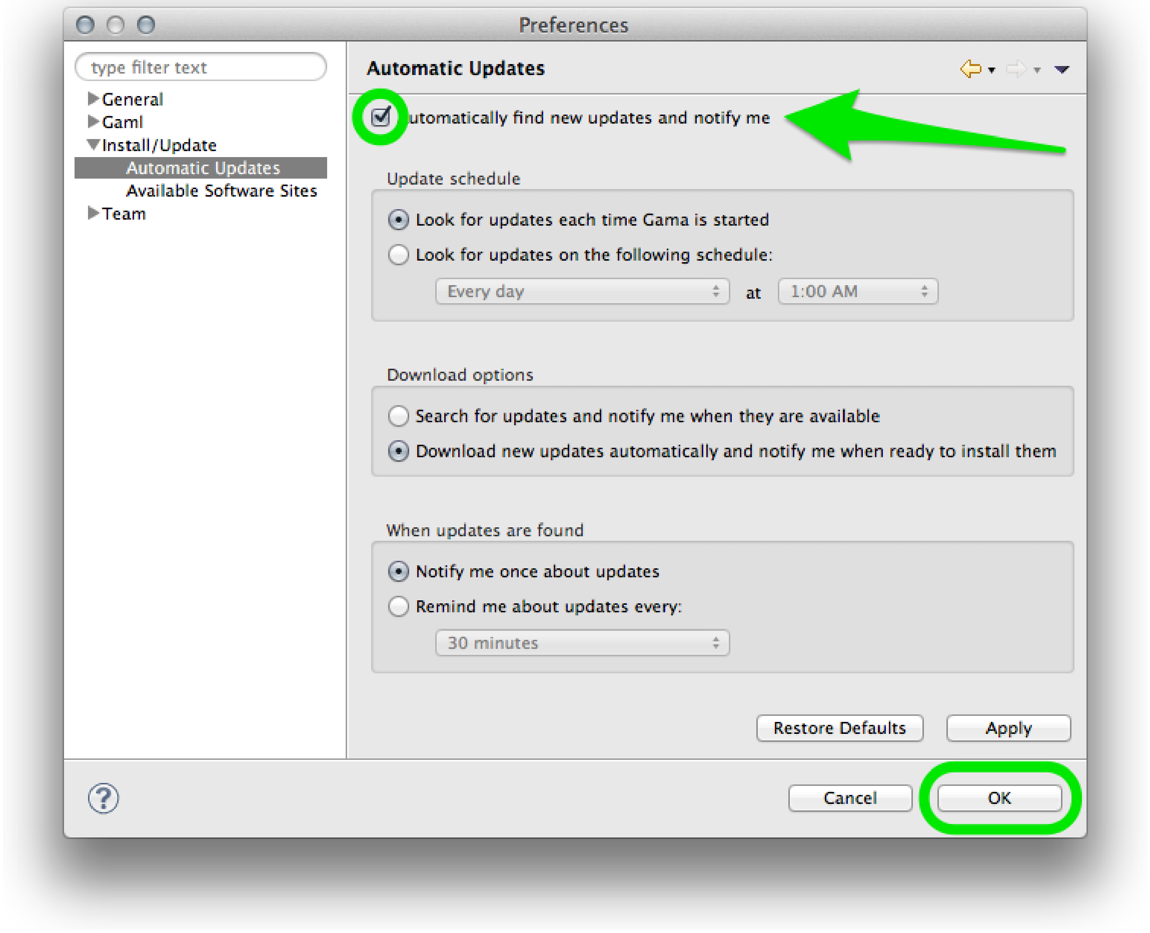
Startup

Display Welcome page at startup: true

Runtime

Default step for delay slider (in sec.): 0.01
Auto-run experiments when they are launched: false
Ask to close the previous simulation before launching a new one ?: false

Figure 4.6: Button to give access to additional preferences.



Automatic Updates

[✓] utomatically find new updates and notify me

Update schedule

[● Look for updates each time Gama is started]
○ Look for updates on the following schedule:
Every day at 1:00 AM

Download options

○ Search for updates and notify me when they are available
[● Download new updates automatically and notify me when ready to install them]

When updates are found

[● Notify me once about updates]
○ Remind me about updates every:
30 minutes

Figure 4.7: Check for automatic update.

Chapter 5

Installing Plugins

Besides the plugins delivered by the developers of the GAMA platform, which can be installed and updated as explained here, there are a number of additional plugins that can be installed to add new functionalities to GAMA or enhance the existing ones. GAMA being based on Eclipse, a number of plugins developed for Eclipse are then available (a complete listing of Eclipse plugins can be found in the so-called Eclipse MarketPlace).

There are, however, three important restrictions:

1. The current version of GAMA is based on Eclipse Juno (version number 3.8.2), which excludes de facto all the plugins targeting solely the 4.3 (Kepler) or 4.4 (Luna) versions of Eclipse. These will refuse to install anyway.
2. The Eclipse foundations in GAMA are only a subset of the complete Eclipse platform, and a number of libraries or frameworks (for example the Java Development Toolkit) are not (and will never be) installed in GAMA. So plugins relying on their existence will refuse to install as well.
3. Some components of GAMA rely on a specific version of other plugins and will refuse to work with other versions, essentially because their compatibility will not be ensured anymore. For instance, the parser and validator of the GAML language in GAMA 1.6.1 require XText v. 2.4.1 to be installed (and neither XText 2.5.4 nor XText 2.3 will satisfy this dependency).

With these restrictions in mind, it is however possible to install interesting additional plugins. We propose here a list of some of these plugins (known to work with GAMA), but feel free to either add a comment if you have tested plugins not listed here or create an issue if a plugin does not work, in order for us to see what the requirements to make it work are and how we can satisfy them (or not) in GAMA.

Table of contents

Installation

Installing new plugins is a process identical to the one described when updating GAMA, with one exception: the *update site* to enter is normally provided by the vendor of the additional plugin and must be entered instead of GAMA's one in the dialog. Let us suppose, for instance, that we want to install a RSS feed reader available on this site. An excerpt from the page reads that :

> All plugins are installed with the standard update manager of Eclipse. It will guide you through the installation process and also eases keeping your plugins up-to-date. Just add the update site: http://www.junginger.biz/eclipse/

So we just have to follow these instructions, which leads us to the following dialog, in which we select "RSS view" and click "Next".

The initial dialog is followed by two other ones, a first to report that the plugin satisfies all the dependencies, a second to ask the user to accept the license agreement.

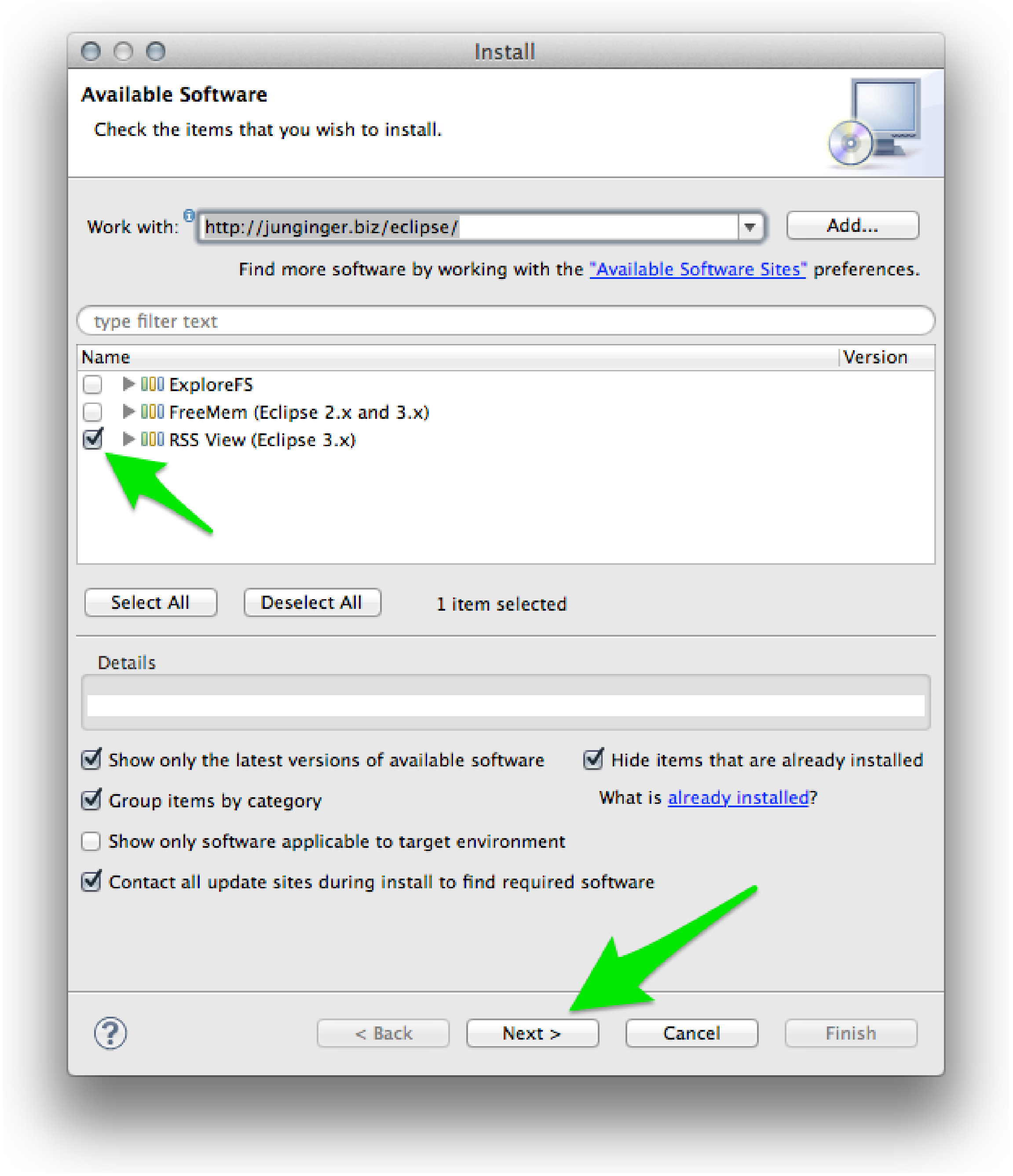
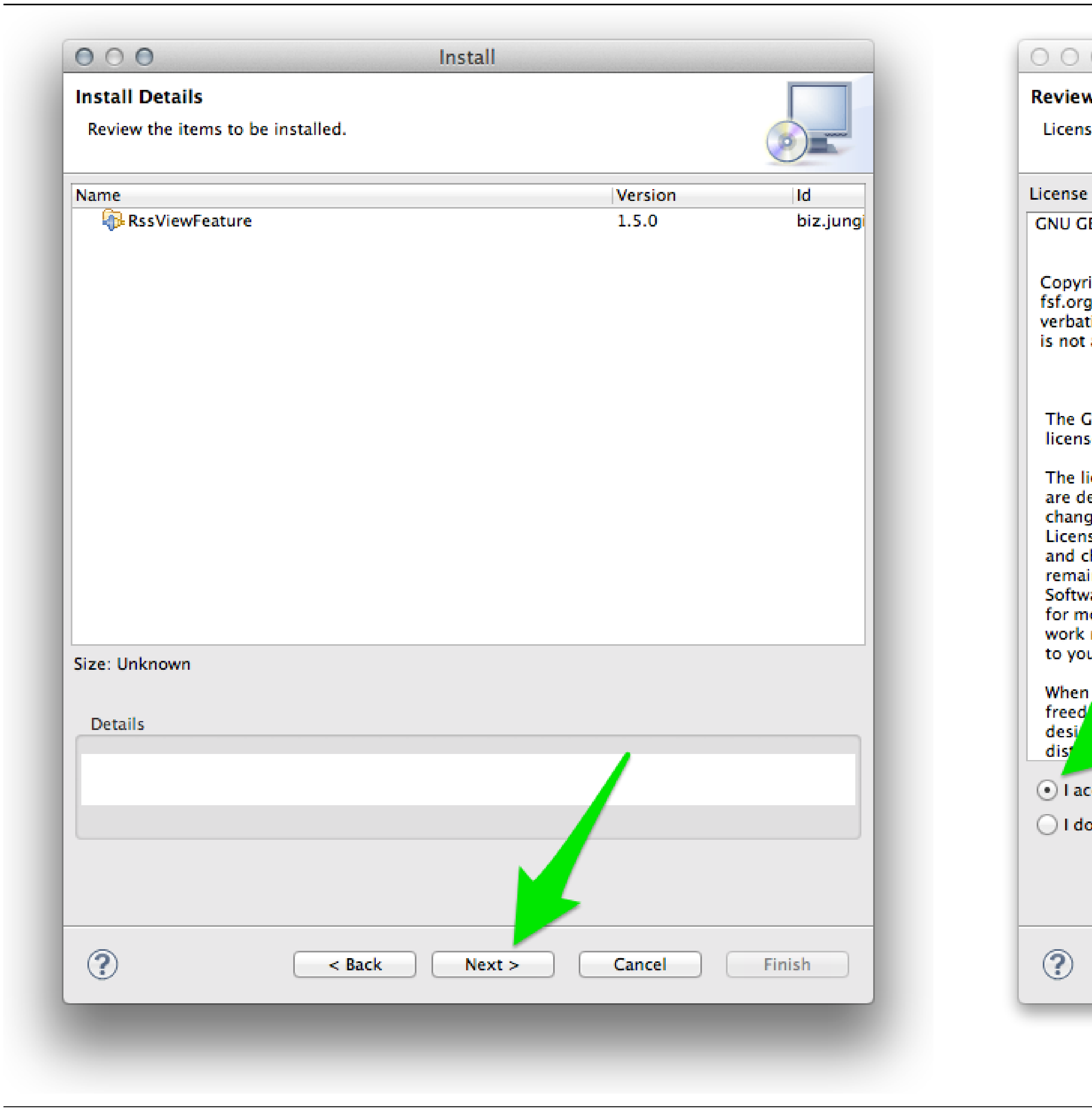

Install
Available Software
Check the items that you wish to install.
Work with:
http://junginger.biz/eclipse/
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
ExploreFS
FreeMem (Eclipse 2.x and 3.x)
RSS View (Eclipse 3.x)
Select All
Deselect All
1 item selected
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish

Figure 5.1: images/dialog_install_plugins.png



Once we dismiss the warning that the plugin is not signed and accept to restart GAMA, we can test the new plugin by going to the "Views" menu.

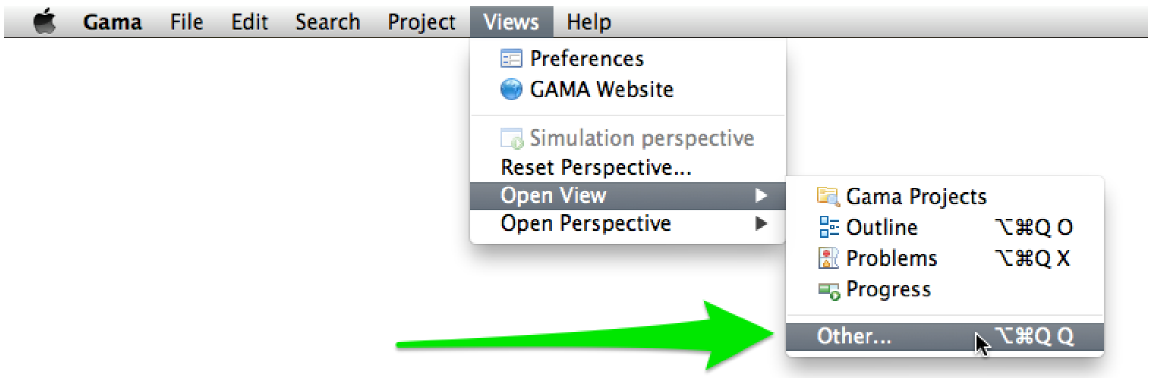


Figure 5.2: images/menu_other_views.png

The new RSS view is available in the list of views that can be displayed in GAMA.

And we can enjoy (after setting some preferences available in its local menu) monitoring the Issues of GAMA from within GAMA itself !

Selected Plugins

In addition to the RSS reader described above, below is a list of plugins that have been tested to work with GAMA. There are many others so take the time to explore them !

Overview

- A very useful plugin for working with large model files. It renders an overview of the file in a separate view (with a user selectable font size), allowing to know where the edition takes place, but also to navigate very quickly and efficiently to different places in the model.
- Update site: http://sandipchitaleseclipseplugins.googlecode.com/svn/trunk/text.overview.updatesite/site.x
- After installing the plugin, an error might happen when closing GAMA. It is harmless. After restarting GAMA, go to Views > Open View > Others... > Overview >.

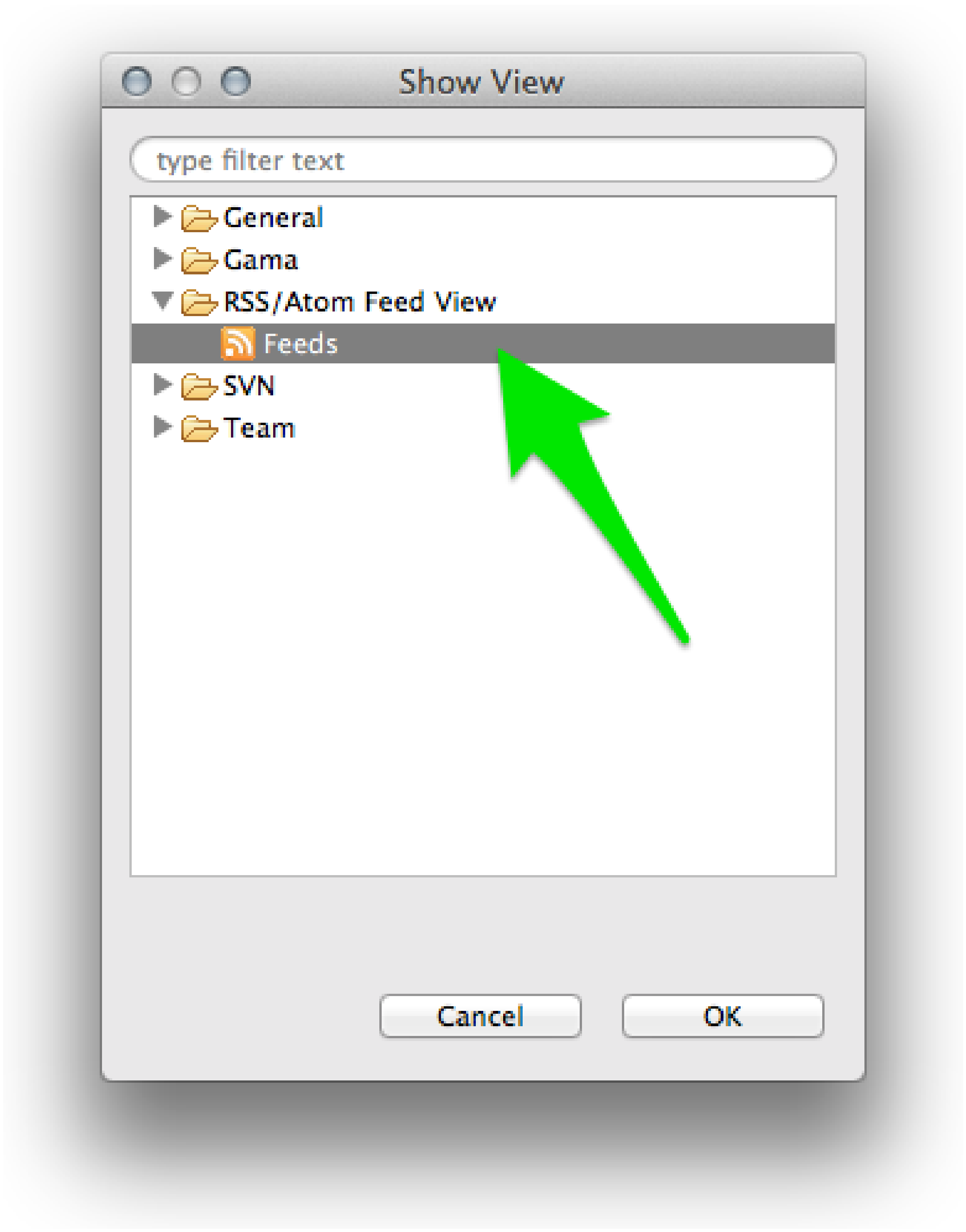
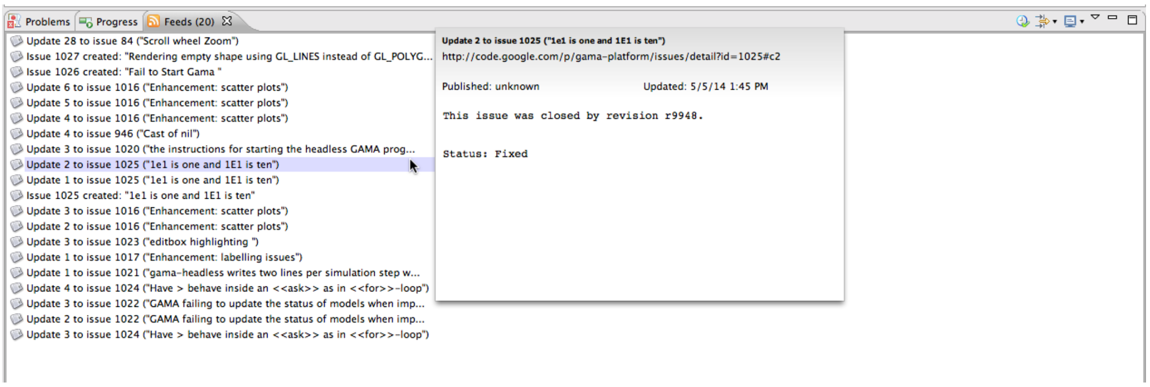


Figure 5.3: images/dialog_show_view.png



```
Update 28 to issue 84 ("Scroll wheel Zoom")
Issue 1027 created: "Rendering empty shape using GL_LINES instead of GL_POLYG...
Issue 1026 created: "Fail to Start Gama "
Update 6 to issue 1016 ("Enhancement: scatter plots")
Update 5 to issue 1016 ("Enhancement: scatter plots")
Update 4 to issue 1016 ("Enhancement: scatter plots")
Update 4 to issue 946 ("Cast of nil")
Update 3 to issue 1020 ("the instructions for starting the headless GAMA prog...
Update 2 to issue 1025 ("1e1 is one and 1E1 is ten")
Update 1 to issue 1025 ("1e1 is one and 1E1 is ten")
Issue 1025 created: "1e1 is one and 1E1 is ten"
Update 3 to issue 1016 ("Enhancement: scatter plots")
Update 2 to issue 1016 ("Enhancement: scatter plots")
Update 3 to issue 1023 ("editbox highlighting ")
Update 1 to issue 1017 ("Enhancement: labelling issues")
Update 1 to issue 1021 ("gama-headless writes two lines per simulation step w...
Update 4 to issue 1024 ("Have > behave inside an <<ask>> as in <<for>>-loop")
Update 3 to issue 1022 ("GAMA failing to update the status of models when imp...
Update 2 to issue 1022 ("GAMA failing to update the status of models when imp...
Update 3 to issue 1024 ("Have > behave inside an <<ask>> as in <<for>>-loop")
```

```
Update 2 to issue 1025 ("1e1 is one and 1E1 is ten")
http://code.google.com/p/gama-platform/issues/detail?id=1025#c2

Published: unknown                Updated: 5/5/14 1:45 PM

This issue was closed by revision r9948.

Status: Fixed
```

Figure 5.4: images/feed_working.png

Git

- Git is a version control system (like CVS or SVN, extensively used in GAMA) http://git-scm.com/. Free sharing space are provided on website such as GitHub or Google Code among others. Installing Git allows to share or gather models that are available in Git repositories.
- Update site (general): `http://download.eclipse.org/releases/juno/`
- Select the two following plugins:
  - Eclipse EGit
  - Git Team Provider Core

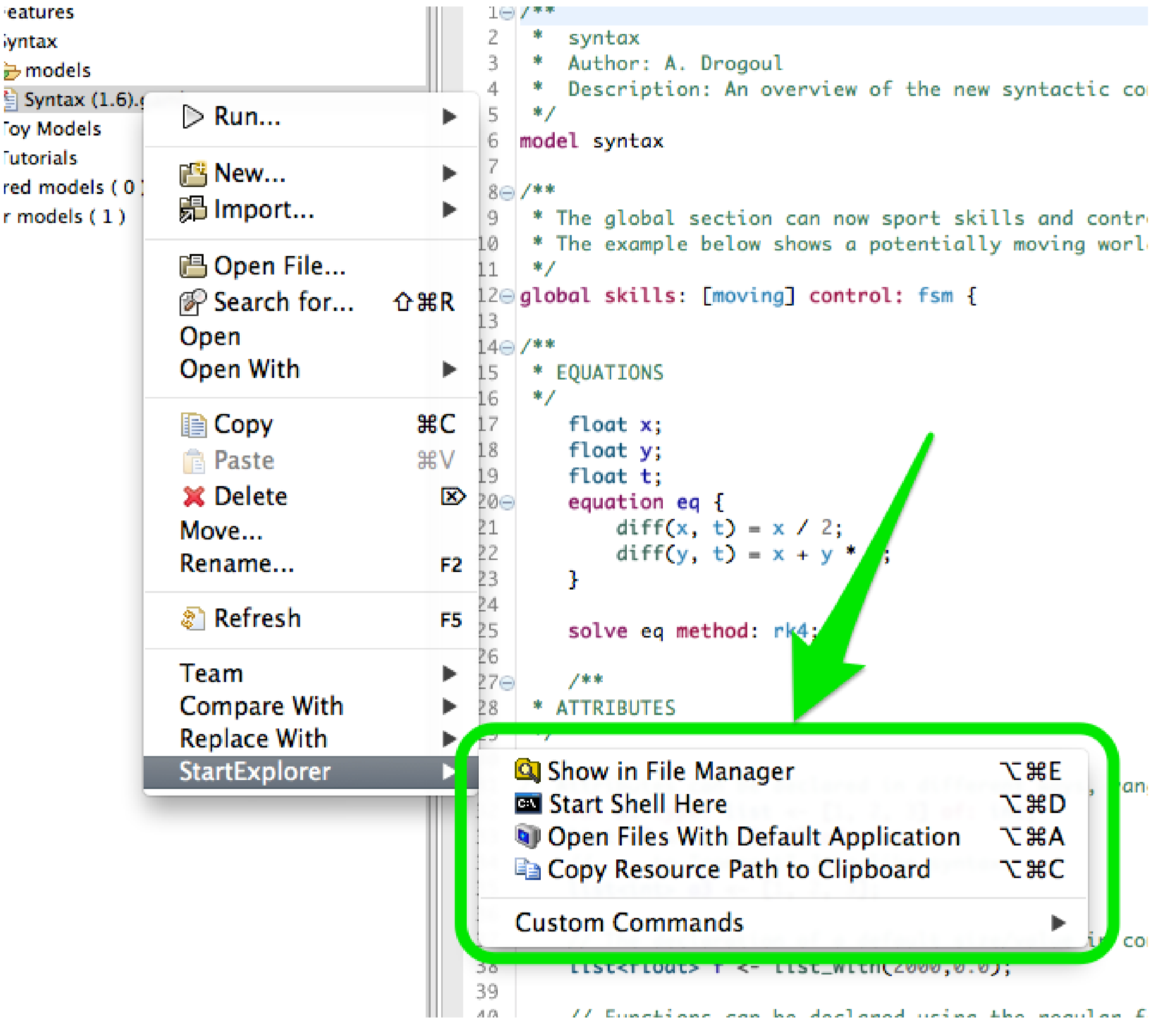
CKEditor

- CKEditor is a lightweight and powerful web-based editor, perfect for almost WYSIWYG edition of HTML files. It can be installed, directly in GAMA, in order to edit .html, .htm, .xml, .svg, etc. files directly without leaving the platform. No other dependencies are required. A must !
- Update site: `http://kosz.bitbucket.org/eclipse-ckeditor/update-site`

Startexplorer

- A nice utility that allows the user to select files, folders or projects in the Navigator and open them in the filesystem (either the UI Explorer, Finder, whatever, or in a terminal).

- Update site: `http://basti1302.github.com/startexplorer/update/`



```
/**
 *  syntax
 *  Author: A. Drogoul
 *  Description: An overview of the new syntactic co
 */
model syntax

/**
 * The global section can now sport skills and contr
 * The example below shows a potentially moving worl
 */
global skills: [moving] control: fsm {

/**
 * EQUATIONS
 */
    float x;
    float y;
    float t;
    equation eq {
        diff(x, t) = x / 2;
        diff(y, t) = x + y *  ;
    }

    solve eq method: rk4;

    /**
 * ATTRIBUTES
```

Figure 5.5: images/start_explorer.png

Pathtools

- Same purpose as StartExplorer, but much more complete, and additionally offers the possibility to add new commands to handle files (open them in specific editors, execute external programs on them, etc.). Very nice and professional. Works flawlessly in

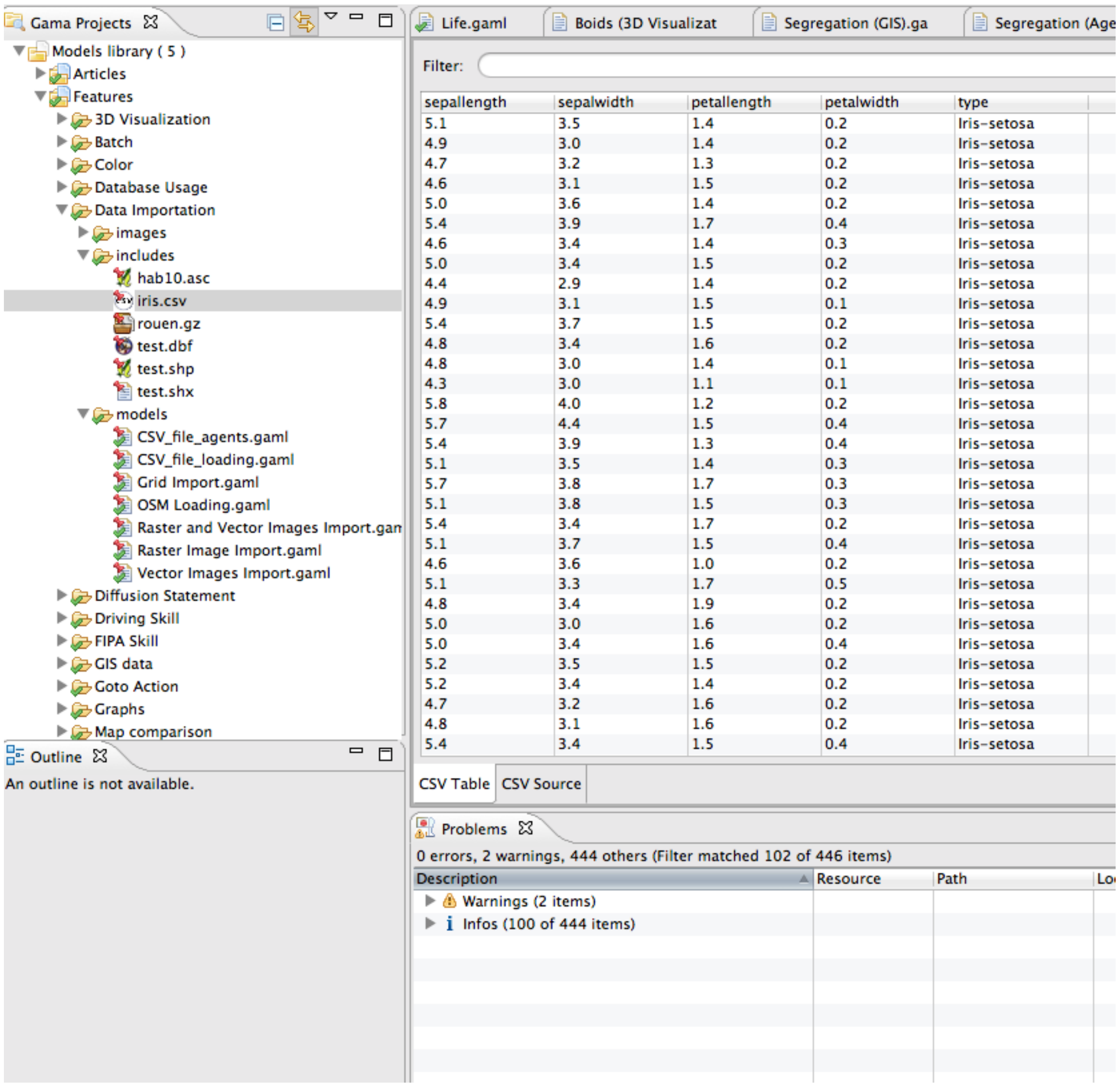
GAMA except that contributions to the toolbar are not accepted (so you have to rely on the commands present in the Navigator pop-up menu).
- Update site: `http://pathtools.googlecode.com/svn/trunk/PathToolsUpdateSite/site.xml`
- Website: `https://pathtools.googlecode.com`

CSV Edit

- An editor for CSV files. Quite handy if you do not want to launch Excel every time you need to inspect or change the CSV data files used in models.
- Update site: `http://csvedit.googlecode.com/svn/trunk/csvedit.update`

Quickimage

- A lightweight viewer of images, which can be useful when several images are used in a model.
- Update site: `http://psnet.nu/eclipse/updates`



Gama Projects

```
▼ Models library ( 5 )
  ▶ Articles
  ▼ Features
    ▶ 3D Visualization
    ▶ Batch
    ▶ Color
    ▶ Database Usage
    ▼ Data Importation
      ▶ images
      ▼ includes
          hab10.asc
          iris.csv
          rouen.gz
          test.dbf
          test.shp
          test.shx
      ▼ models
          CSV_file_agents.gaml
          CSV_file_loading.gaml
          Grid Import.gaml
          OSM Loading.gaml
          Raster and Vector Images Import.gam
          Raster Image Import.gaml
          Vector Images Import.gaml
    ▶ Diffusion Statement
    ▶ Driving Skill
    ▶ FIPA Skill
    ▶ GIS data
    ▶ Goto Action
    ▶ Graphs
    ▶ Map comparison
```

Outline

An outline is not available.

Life.gaml | Boids (3D Visualizat | Segregation (GIS).ga | Segregation (Age

Filter:

sepallength	sepalwidth	petallength	petalwidth	type
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5.0	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
4.8	3.0	1.4	0.1	Iris-setosa
4.3	3.0	1.1	0.1	Iris-setosa
5.8	4.0	1.2	0.2	Iris-setosa
5.7	4.4	1.5	0.4	Iris-setosa
5.4	3.9	1.3	0.4	Iris-setosa
5.1	3.5	1.4	0.3	Iris-setosa
5.7	3.8	1.7	0.3	Iris-setosa
5.1	3.8	1.5	0.3	Iris-setosa
5.4	3.4	1.7	0.2	Iris-setosa
5.1	3.7	1.5	0.4	Iris-setosa
4.6	3.6	1.0	0.2	Iris-setosa
5.1	3.3	1.7	0.5	Iris-setosa
4.8	3.4	1.9	0.2	Iris-setosa
5.0	3.0	1.6	0.2	Iris-setosa
5.0	3.4	1.6	0.4	Iris-setosa
5.2	3.5	1.5	0.2	Iris-setosa
5.2	3.4	1.4	0.2	Iris-setosa
4.7	3.2	1.6	0.2	Iris-setosa
4.8	3.1	1.6	0.2	Iris-setosa
5.4	3.4	1.5	0.4	Iris-setosa

CSV Table | CSV Source

Problems

0 errors, 2 warnings, 444 others (Filter matched 102 of 446 items)

Description	Resource	Path	Lo
▶ ⚠ Warnings (2 items)			
▶ i Infos (100 of 444 items)			

Figure 5.6: images/csv_edit.png

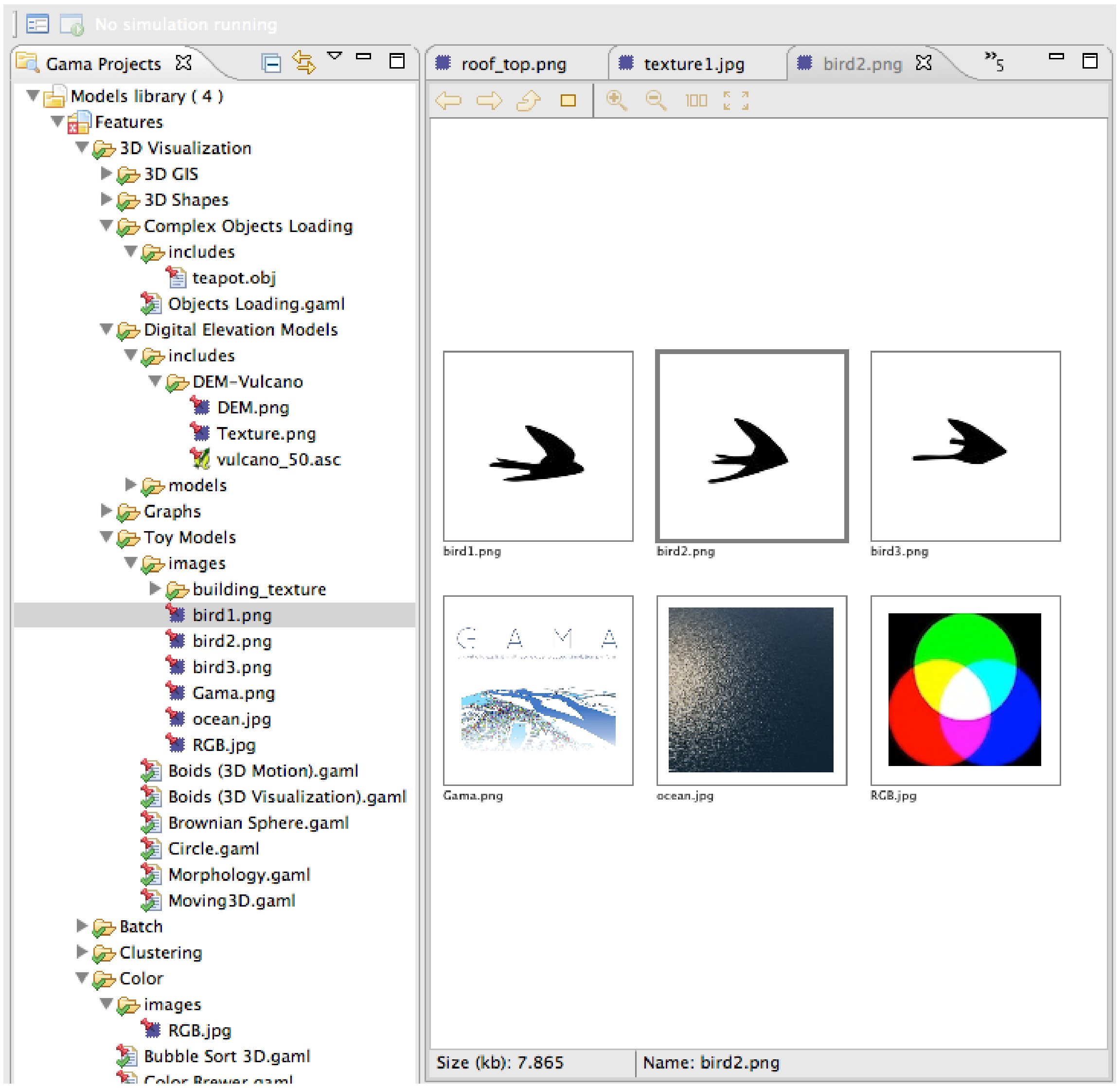


Figure 5.7: images/quick_image.png

Chapter 6

Troubleshooting

This page exposes some of the most common problems a user may encounter when running GAMA — and offers advices and workarounds for them. It will be regularly enriched with new contents. Note also that the Issues section of the website might contain precious information on crashes and bugs encountered by other users. If neither the workarounds described here nor the solutions provided by other users allow to solve your particular problem, please submit a new issue report to the developers.

Table of contents

On Ubuntu (& Linux Systems)

Workaround if GAMA crashes when displaying web contents

In case GAMA crashes whenever trying to display a web page or the pop-up online documentation, you may try to edit the file Gama.ini and add the line `-Dorg.eclipse.swt.browser.DefaultType=mozilla` to it. This workaround is described here: http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=705420 and in Issue 700 (on Google Code).

Workaround if GAMA does not display the menus (the 'Edit' menu is the only one working)

If, when selecting a menu, nothing happens (or, in the case of the 'Agents' menu, all population submenus appear empty), it is likely that you have run into this issue: https://bugs.eclipse.org/bugs/show_bug.cgi?id=330563. The only workaround known is to launch GAMA from the command line (or from a shell script) after having told Ubuntu to attach its menu back to its main window. For example (if you are in the directory where the "Gama" executable is present):

```
export UBUNTU_MENUPROXY=0
./Gama
```

No fix can be provided from the GAMA side for the moment.

On Windows

No common trouble…

On MacOS X

Workaround in case of glitches in the UI

The only problems reported so far on MacOS X (from Lion to Yosemite) concern visual glitches in the UI and problems with displays, either not showing or crashing the JVM. Most (all ?) of these problems are usually related to the fact that GAMA does not run under the correct version of Java Virtual Machine. In that case, follow these instructions to install the correct version.

Workaround in case of corrupted icons in menus under El Capitan

For some particular configurations (in particular some particular graphic cards), the icons of the menus (e.g. Edit menu) may be corrupted. This bug is documented for all RCP products under El Capitan. See these references: https://bugs.eclipse.org/bugs/show_bug.cgi?id=479590 https://trac.filezilla-project.org/ticket/10669

These is nothing we can do now except using the workaround that consists in switching the language of the OS to English (in System Preferences, Language & Region).

Memory problems

The most common causes of problems when running GAMA are memory problems. Depending on your activities, on the size of the models you are editing, on the size of the experiments you are running, etc., you have a chance to require more memory than what is currently allocated to GAMA. A typical GAMA installation will need between 40 and 200MB of memory to run "normally" and launch small models. Memory problems are easy to detect: on the bottom right corner of its window, GAMA will always display the status of the current memory. The first number represents the memory currently used (in MB), the second (always larger) the memory currently allocated by the JVM. And the little trash icon allows to "garbage collect" the memory still used by agents that are not used anymore (if any). If GAMA appears to hang or crash and if you can see that the two numbers are very close, it means that the memory required by GAMA exceeds the memory allocated.

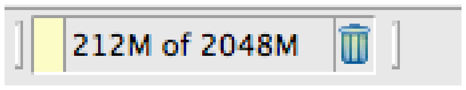
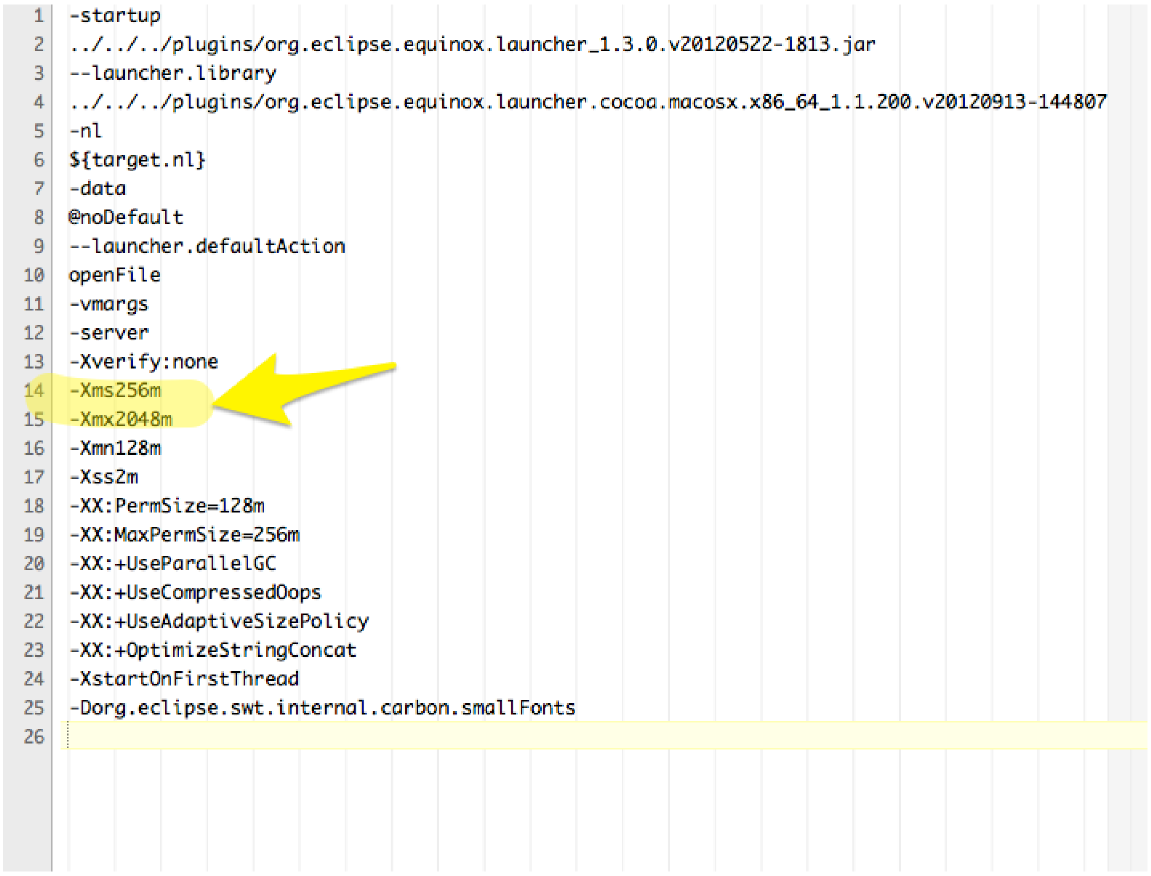


Figure 6.1: images/memory_status.png

There are two ways to circumvent this problem: the first one is to increase the memory allocated to GAMA by the Java Virtual Machine. The second, detailed on this page is to try to optimize your models to reduce their memory footprint at runtime. To increase the memory allocated, first locate the file called `Gama.ini`. On Windows and Ubuntu, it is located next to the executable. On MacOS X, you have to right-click on `Gama.app`, choose “Display Package Contents...”, and you will find `Gama.ini` in `Contents/MacOS`. This file typically looks like the following (some options/keywords may vary depending on the system), and we are interested in two JVM arguments:



```
-startup
../../../plugins/org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar
--launcher.library
../../../plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.200.v20120913-144807
-nl
${target.nl}
-data
@noDefault
--launcher.defaultAction
openFile
-vmargs
-server
-Xverify:none
-Xms256m
-Xmx2048m
-Xmn128m
-Xss2m
-XX:PermSize=128m
-XX:MaxPermSize=256m
-XX:+UseParallelGC
-XX:+UseCompressedOops
-XX:+UseAdaptiveSizePolicy
-XX:+OptimizeStringConcat
-XstartOnFirstThread
-Dorg.eclipse.swt.internal.carbon.smallFonts
```

Figure 6.2: images/gama_ini.png

`-Xms` supplies the minimal amount of memory the JVM should allocate to GAMA, `-Xmx` the maximal amount. By changing these values (esp. the second one, of course, for example to 4096M, or 4g), saving the file and relaunching GAMA, you can probably solve your problem. Note that 32 bits versions of GAMA will not accept to run with a value of `-Xmx` greater than 1500M. See here for additional information on these two options.

Submitting an Issue

If you think you have found a new bug/issue in GAMA, it is time to create an issue report here ! Alternatively, you can click the Issues tab on the project site, search if a similar problem has already been reported (and, maybe, solved) and, if not, enter a new issue with as much information as possible: * A complete description of the problem and how it occurred. * The GAMA model or code you are having trouble with. If possible, attach a complete model. * Screenshots or other files that help describe the issue.

Two files may be particularly interesting to attach to your issue: the **configuration details** and the **error log**. Both can be obtained quite easily from within GAMA itself in a few steps. First, click the "About GAMA..." menu item (under the "Gama" menu on MacOS X, "Help" menu on Linux & Windows)

In the dialog that appears, you will find a button called "Installation Details".

Click this button and a new dialog appears with several tabs.

To provide a complete information about the status of your system at the time of the error, you can

(1) copy and paste the text found in the tab "Configuration" into your issue. Although, it is preferable to attach it as a text file (using textEdit, Notepad or Emacs e.g.) as it may be too long for the comment section of the issue form.

(2) click the "View error log" button, which will bring you to the location, in your file system, of a file called "log", which you can then attach to your issue as well.

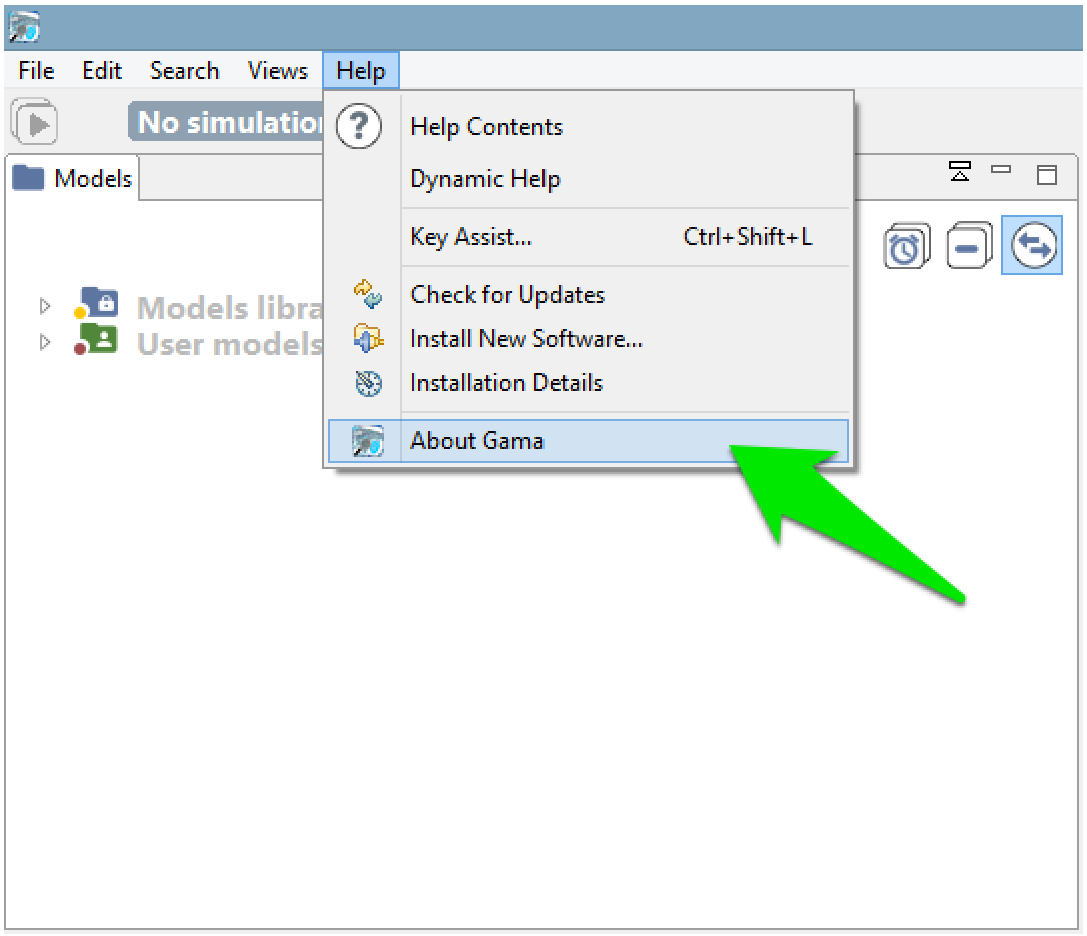
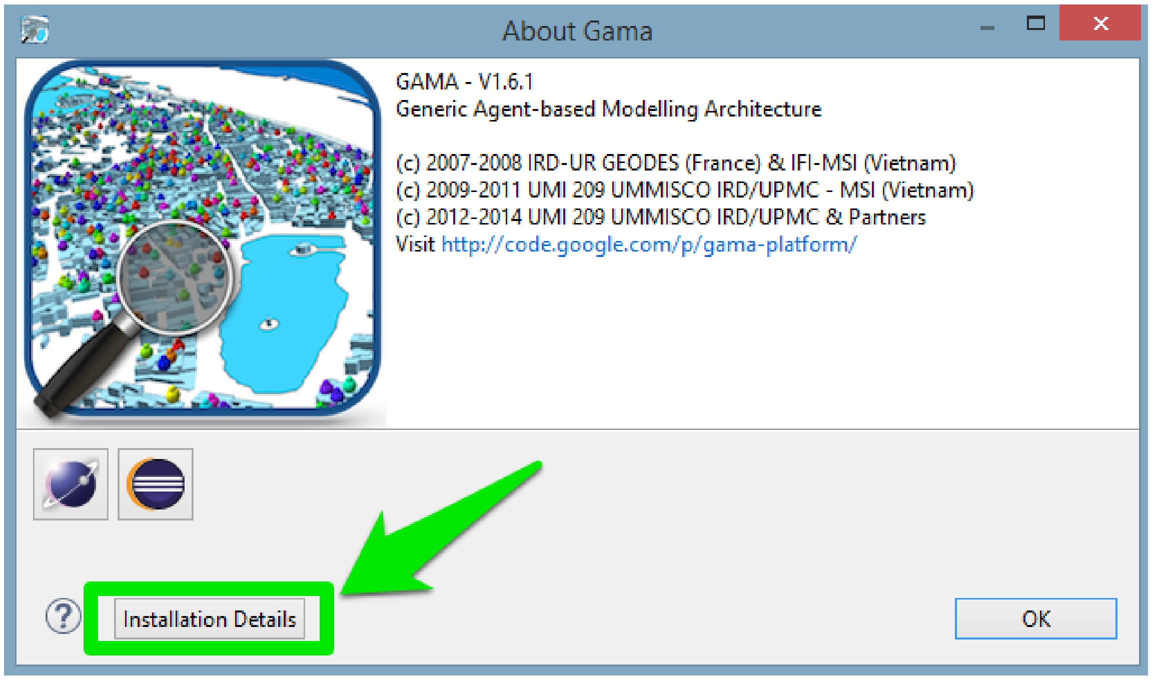


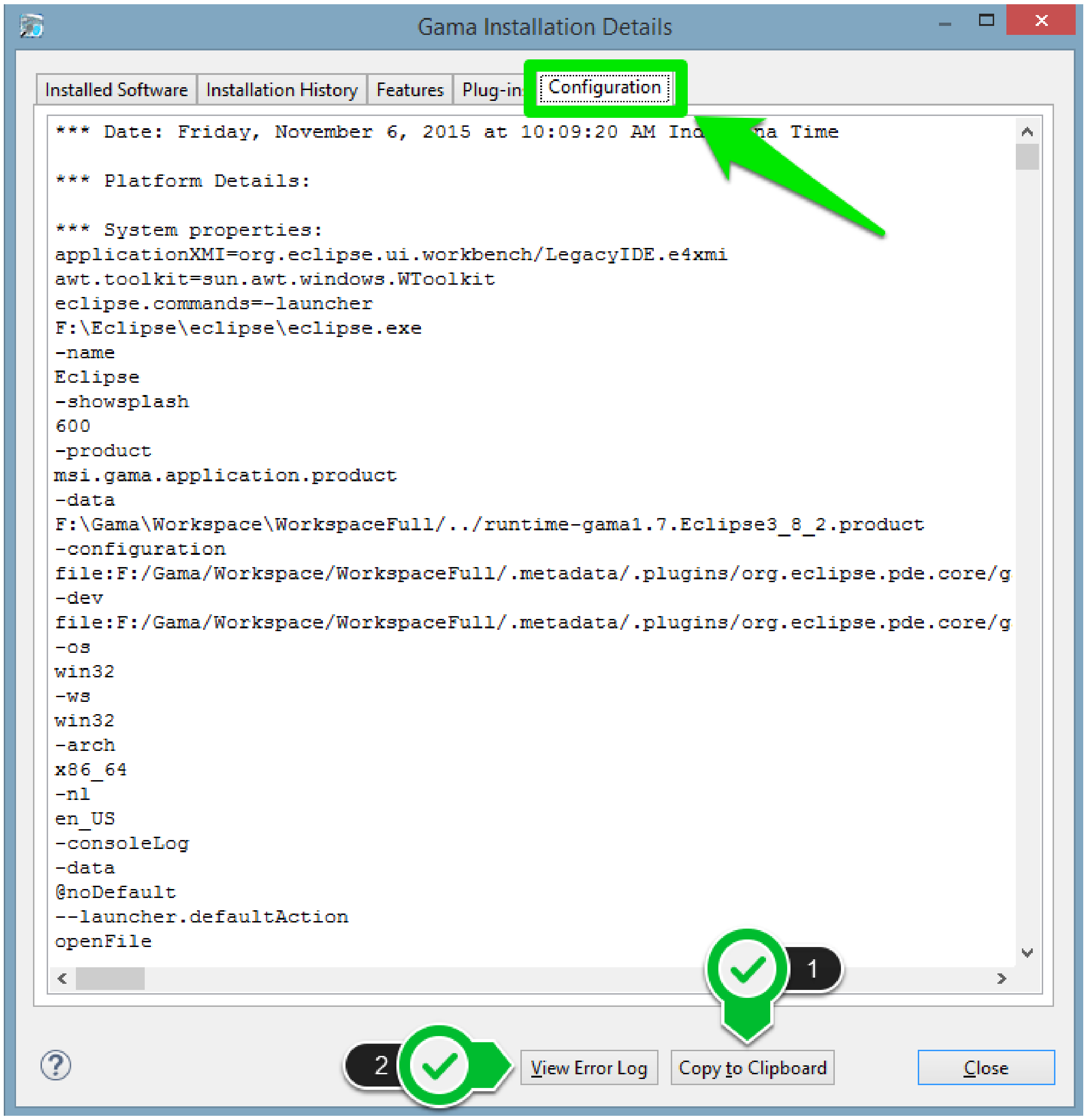
Figure 6.3: images/menu_about_gama.png


About Gama

GAMA - V1.6.1
Generic Agent-based Modelling Architecture

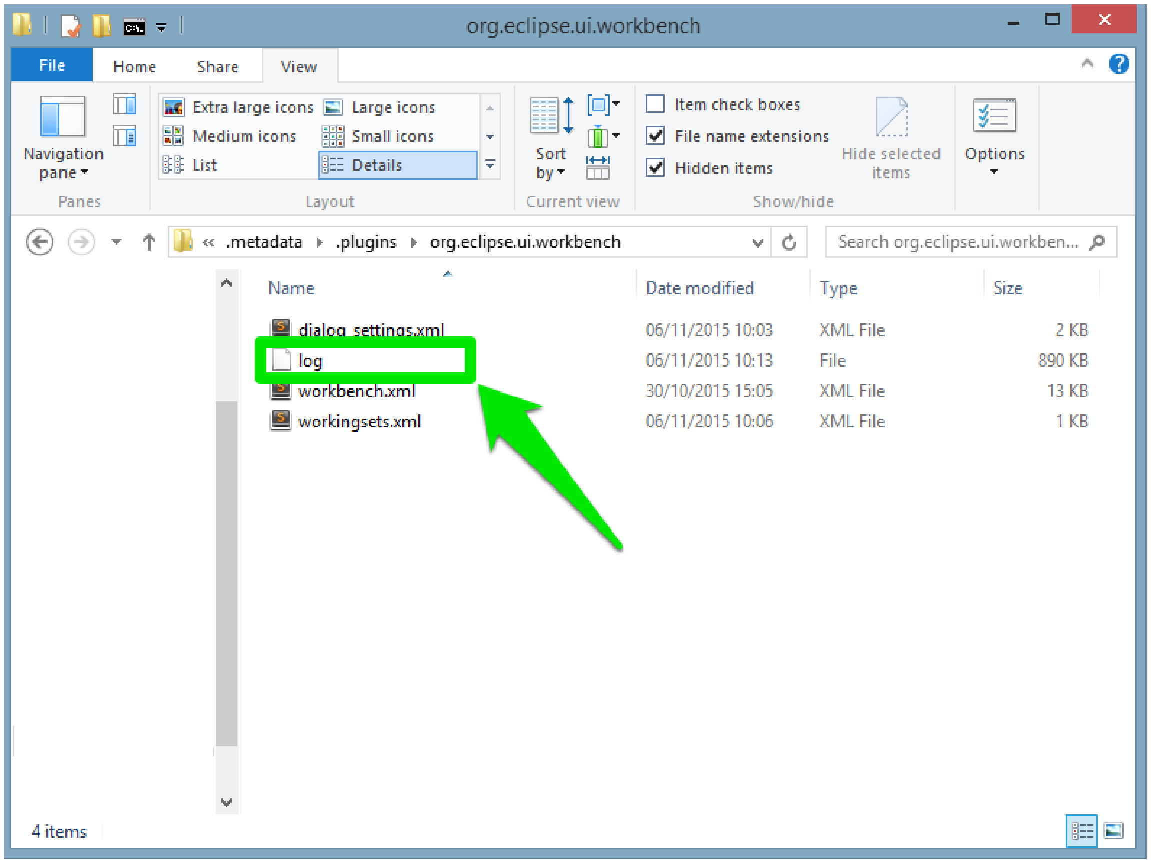
(c) 2007-2008 IRD-UR GEODES (France) & IFI-MSI (Vietnam)
(c) 2009-2011 UMI 209 UMMISCO IRD/UPMC - MSI (Vietnam)
(c) 2012-2014 UMI 209 UMMISCO IRD/UPMC & Partners
Visit http://code.google.com/p/gama-platform/

Figure 6.4: images/dialog_about_gama.png



```
*** Date: Friday, November 6, 2015 at 10:09:20 AM Ind    na Time

*** Platform Details:

*** System properties:
applicationXMI=org.eclipse.ui.workbench/LegacyIDE.e4xmi
awt.toolkit=sun.awt.windows.WToolkit
eclipse.commands=-launcher
F:\Eclipse\eclipse\eclipse.exe
-name
Eclipse
-showsplash
600
-product
msi.gama.application.product
-data
F:\Gama\Workspace\WorkspaceFull/../runtime-gama1.7.Eclipse3_8_2.product
-configuration
file:F:/Gama/Workspace/WorkspaceFull/.metadata/.plugins/org.eclipse.pde.core/g
-dev
file:F:/Gama/Workspace/WorkspaceFull/.metadata/.plugins/org.eclipse.pde.core/g
-os
win32
-ws
win32
-arch
x86_64
-nl
en_US
-consoleLog
-data
@noDefault
--launcher.defaultAction
openFile
```

Figure 6.5: images/dialog_configuration.png



Name	Date modified	Type	Size
dialog_settings.xml	06/11/2015 10:03	XML File	2 KB
log	06/11/2015 10:13	File	890 KB
workbench.xml	30/10/2015 15:05	XML File	13 KB
workingsets.xml	06/11/2015 10:06	XML File	1 KB

Figure 6.6: images/log_file.png

Workspace, Projects and Models

The **workspace** is a directory in which GAMA stores all the current projects on which the user is working, links to other projects, as well as some meta-data like preference settings, current status of the different projects, error markers, and so on.

Except when running in headless mode, **GAMA cannot function without a valid workspace**.

The workspace is organized in 3 **categories**, which are themselves organized into **projects**.

The **projects** present in the **workspace** can be either directly *stored* within it (as sub-directories), which is usually the case when the user creates a new project, or *linked* from it (so the workspace will only contain a link to the directory of the project, supposed to be somewhere in the filesystem or on the network). A same **project** can be linked from different **workspaces**.

GAMA models files are stored in these **projects**, which may contain also other files (called *resources*) necessary for the **models** to function. A project may of course contain several **model files**, especially if they are importing each other, if they represent different views on the same topic, or if they share the same resources.

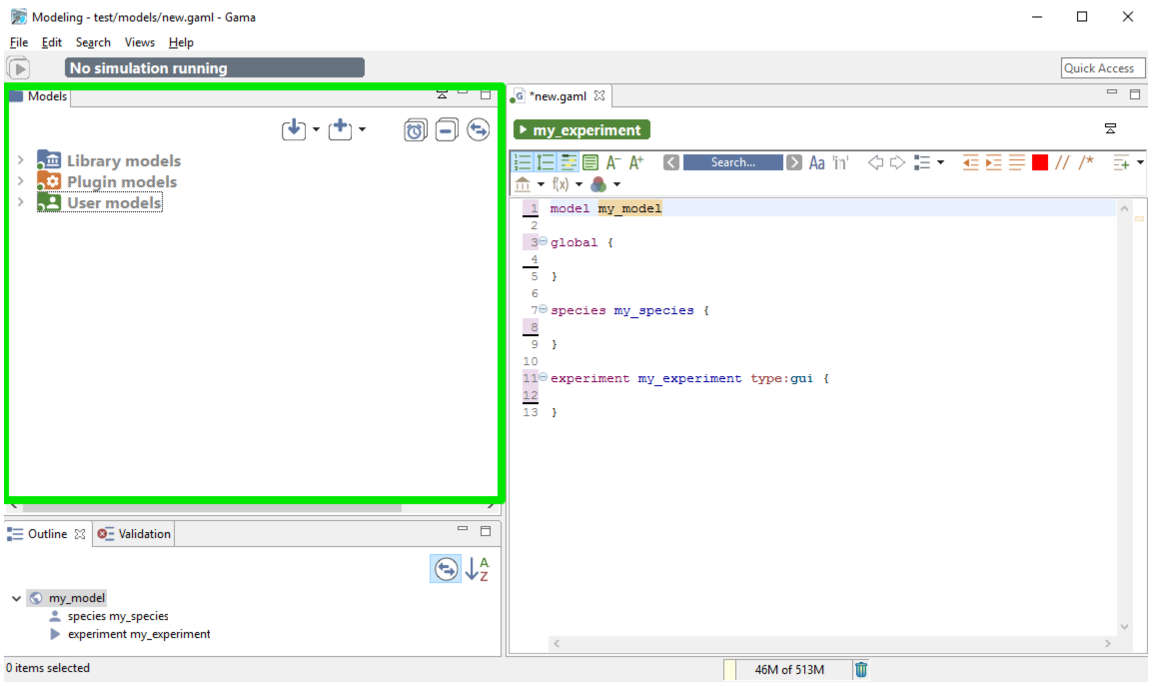
Learning how to navigate in the workspace, how to switch workspace or how to import, export is a necessity to use GAMA correctly. It is the purpose of the following sections.

- 1. Navigating in the Workspace
- 2. Changing Workspace
- 3. Importing Models

Chapter 7

Navigating in the Workspace

All the models that you edit or run using GAMA are accessible from a central location: the *Navigator*, which is always on the left-hand side of the main window and cannot be closed. This view presents the models currently present in (or linked from) your **workspace**.



```
model my_model

global {

}

species my_species {

}

experiment my_experiment type:gui {

}
```

Figure 7.1: images/navigator_first.png

Table of contents

The Different Categories of Models

In the *Navigator*, models are organized in three different categories: *Models library*, *Plugin models*, and *User models*. This organization is purely logical and does not reflect where the models are actually stored in the workspace (or elsewhere). Whatever their actual location, model files need to be stored in **projects**, which may contain also other files (called *resources*) necessary for the models to function. A project may of course contain several model files, especially if they are importing each other, if they represent different models on the same topic, or if they share the same resources.

Models library

This category represents the models that are shipped with each version of GAMA. They do not reside in the workspace, but are considered as *linked* from it. This link is established every time a new workspace is created. Their actual location is within a plugin (msi.gama.models) of the GAMA application. This category contains four main projects in GAMA 1.6.1, which are further refined in folders and sub-folders that contain model files and resources.

It may happen, in some occasions, that the library of models is not synchronized with the version of GAMA that uses your workspace. This is the case if you use different versions of GAMA to work with the same workspace. In that case, it is required that the library be manually updated. This can be done using the “Update library” item in the contextual menu.

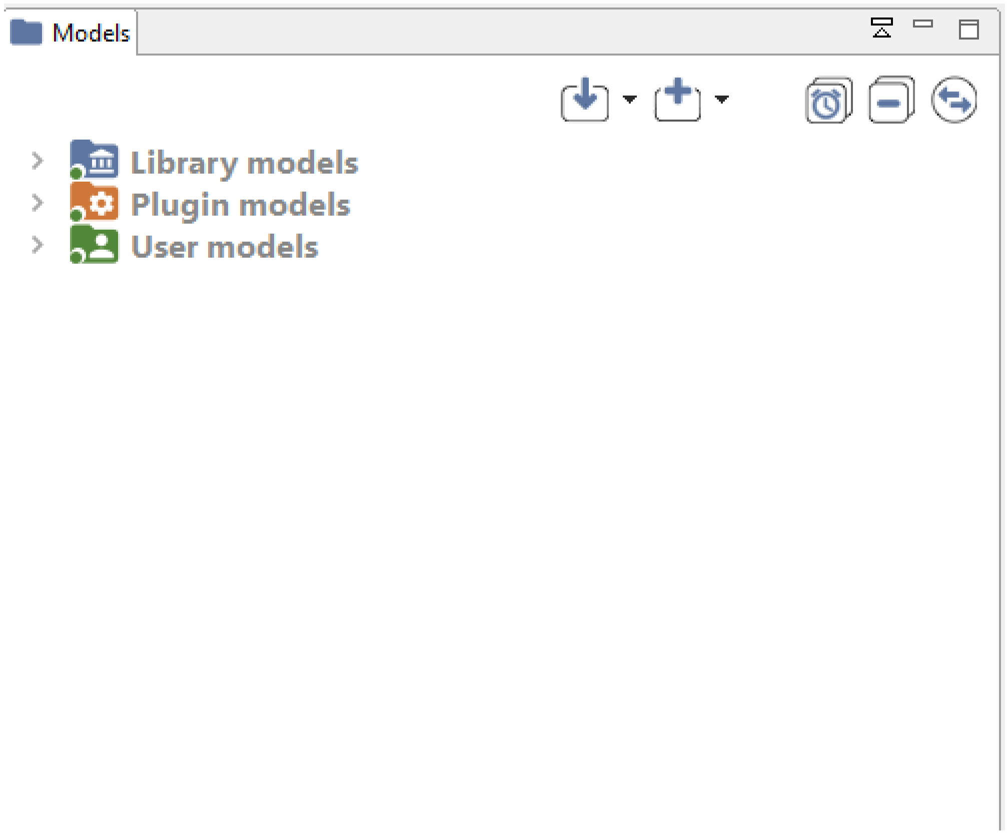


Figure 7.2: images/navigator_3_categories.png

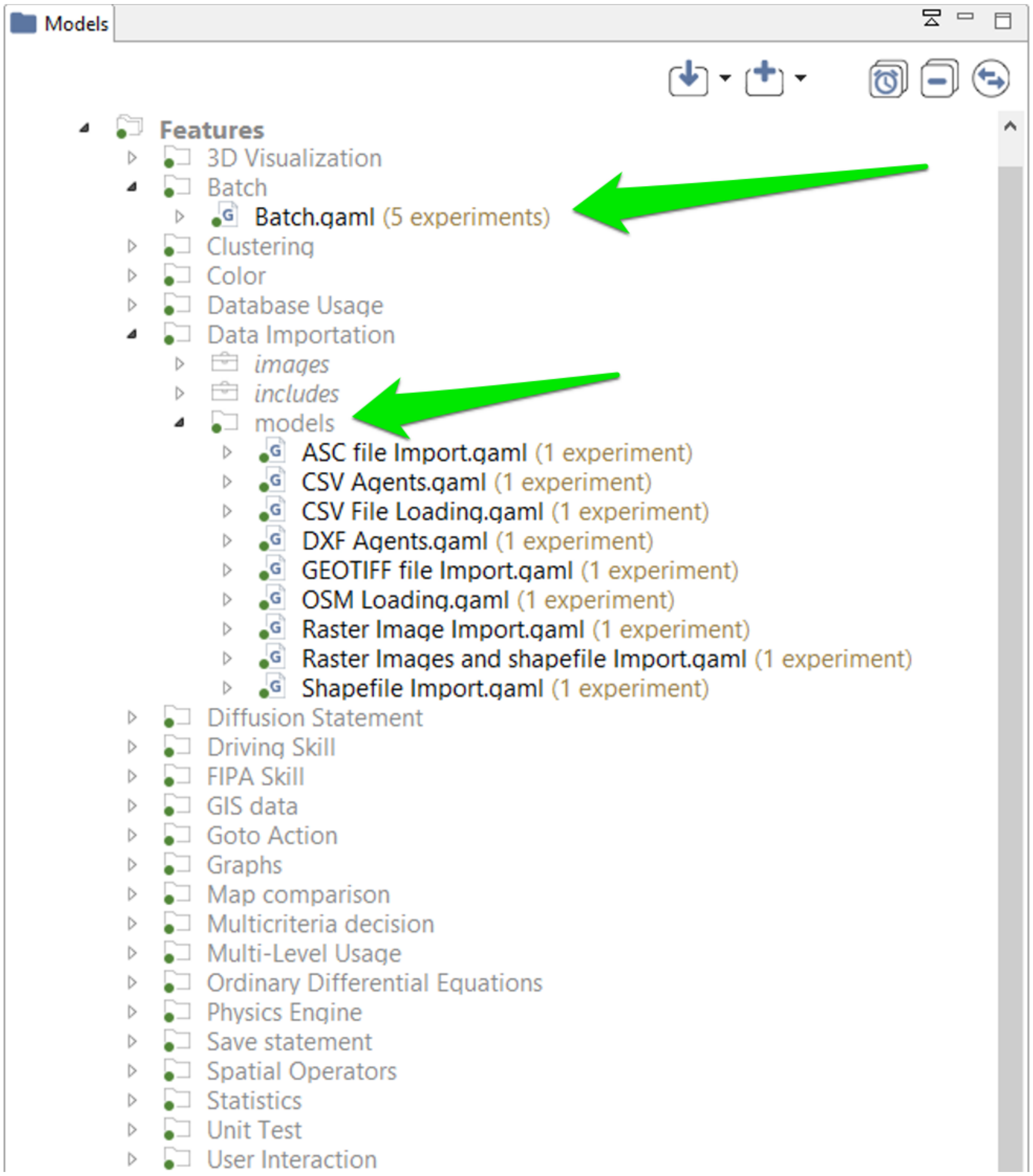

Models
Features
3D Visualization
Batch
Batch.gaml (5 experiments)
Clustering
Color
Database Usage
Data Importation
images
includes
models
ASC file Import.gaml (1 experiment)
CSV Agents.gaml (1 experiment)
CSV File Loading.gaml (1 experiment)
DXF Agents.gaml (1 experiment)
GEOTIFF file Import.gaml (1 experiment)
OSM Loading.gaml (1 experiment)
Raster Image Import.gaml (1 experiment)
Raster Images and shapefile Import.gaml (1 experiment)
Shapefile Import.gaml (1 experiment)
Diffusion Statement
Driving Skill
FIPA Skill
GIS data
Goto Action
Graphs
Map comparison
Multicriteria decision
Multi-Level Usage
Ordinary Differential Equations
Physics Engine
Save statement
Spatial Operators
Statistics
Unit Test
User Interaction

Figure 7.3: images/navigator_library_2_folders_expanded.png

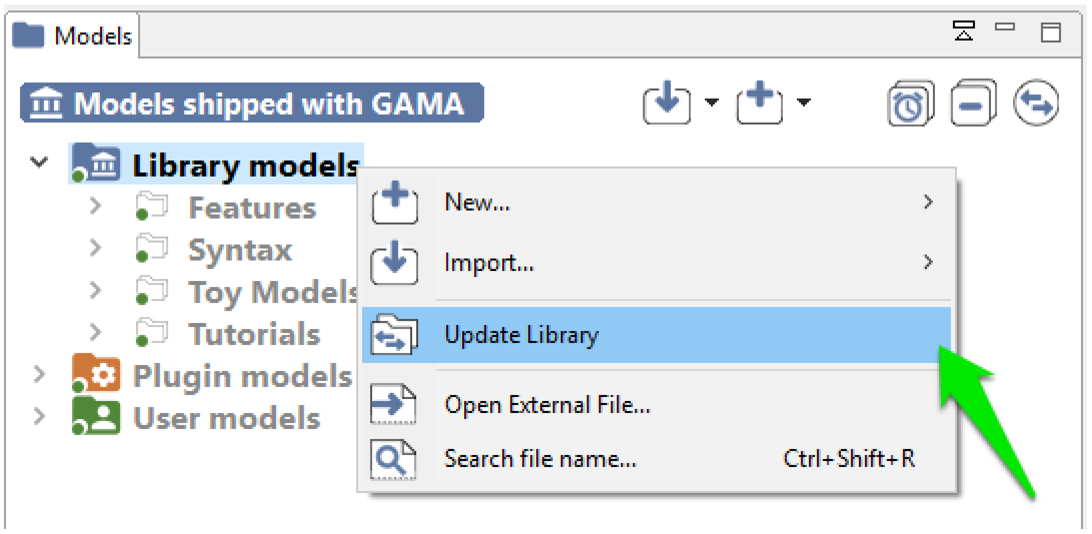
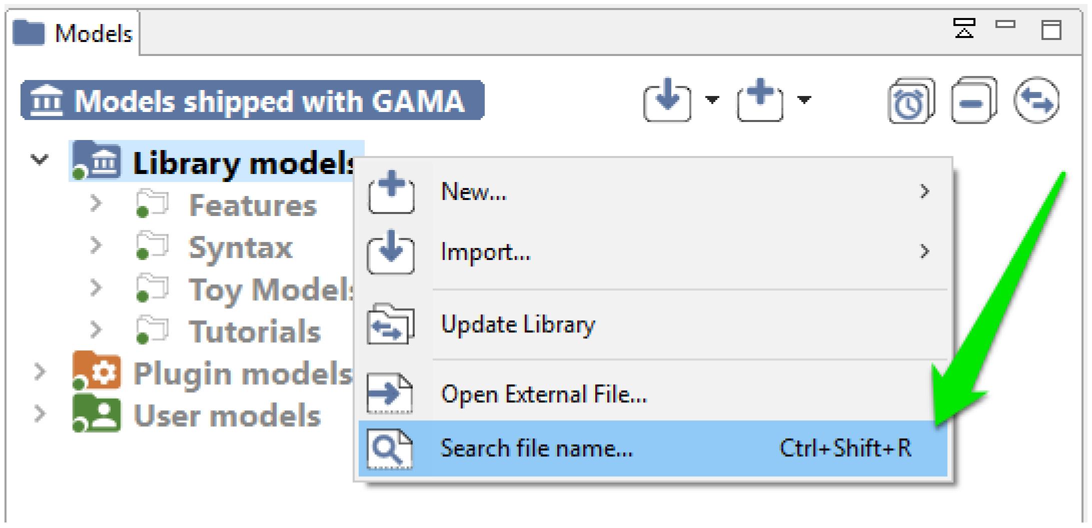
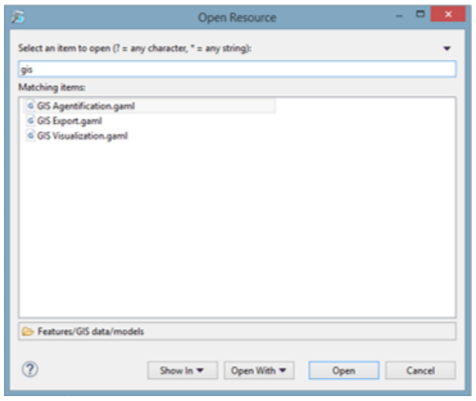


Figure 7.4: images/navigator_update_library.png

To look up for a particular model in the library, users can use the "Search for file" menu item. A search dialog is then displayed, which allows to look for models by their title (for example, models containing "GIS" in the example below).





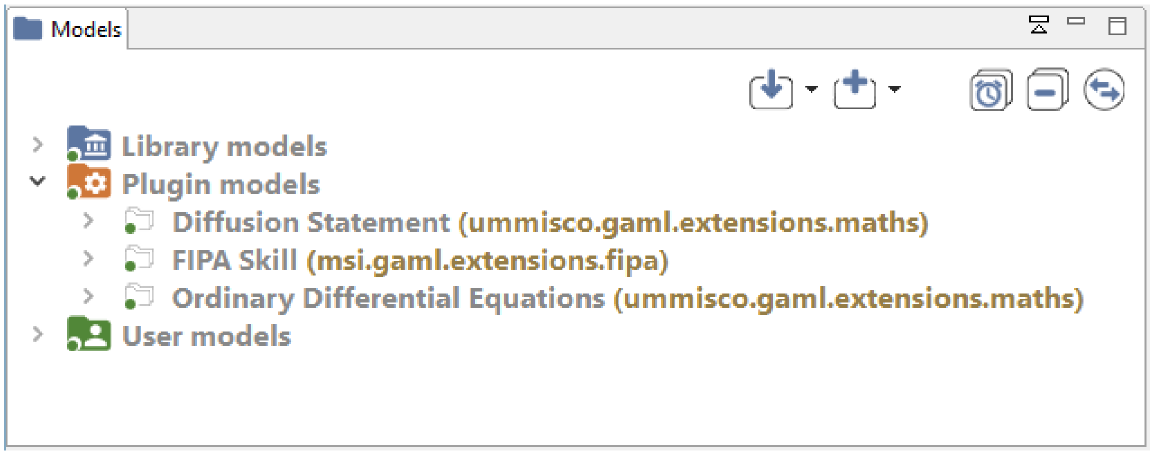
gis

GIS Agentification.gaml
GIS Export.gaml
GIS Visualization.gaml

Features/GIS data/models

Plugin models

This category represents the models that are related to a specific plugin (additional or integrated by default). The corresponding plugin is shown between parenthesis.



Library models
Plugin models
Diffusion Statement (ummisco.gaml.extensions.maths)
FIPA Skill (msi.gaml.extensions.fipa)
Ordinary Differential Equations (ummisco.gaml.extensions.maths)
User models

Figure 7.5: images/navigator_plugin_models.png

For each projects, you can see the list of plugins needed, and a caption to show you if the plugin is actually installed in your GAMA version or not : green if the plugin is installed, red otherwise.

User models

This category regroups all the projects that have been created or imported in the workspace by the user. Each project is actually a folder that resides in the folder of the workspace (so they can be easily located from within the filesystem). Any modification (addition, removal of files…) made to them in the filesystem (or using another application) is immediately reflected in the *Navigator* and vice-versa.

Model files, although it is by no means mandatory, usually reside in a sub-folder of the project called "models".

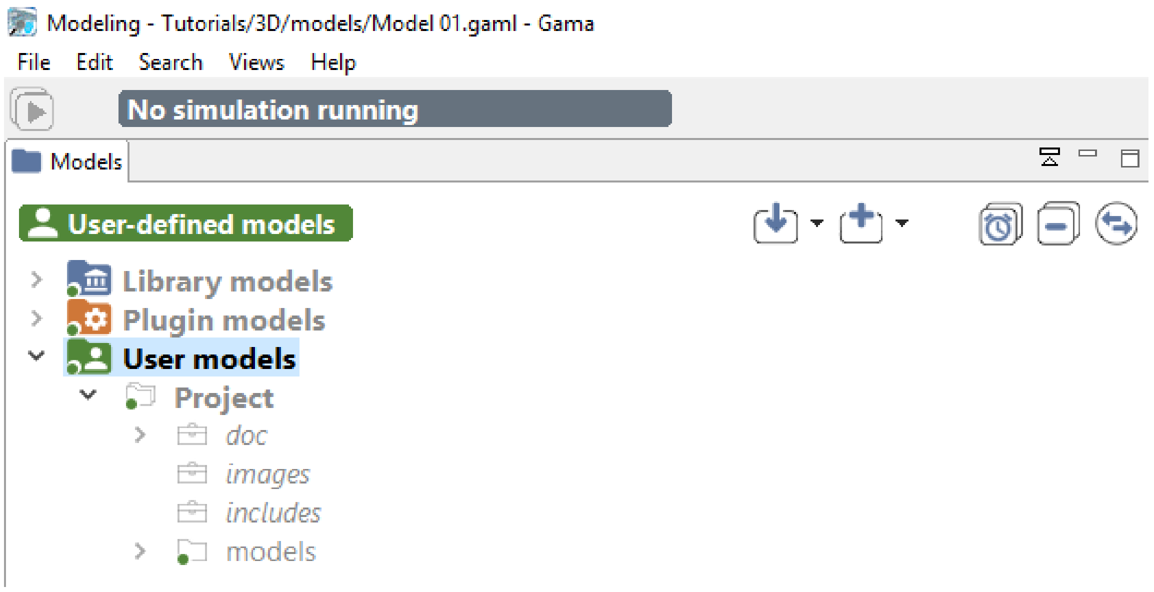


Figure 7.6: images/navigator_user_expanded.png

Inspect Models

Each models is presented as a node in the navigation workspace, including *Experiment* buttons and/or *Requires* node and/or *Uses* node.

- **Experiment button** : Experiment button are present if your model contains experiments (it is usually the case !). To run the corresponding experiment, just click on it. To learn more about running experiments, jump into this section.

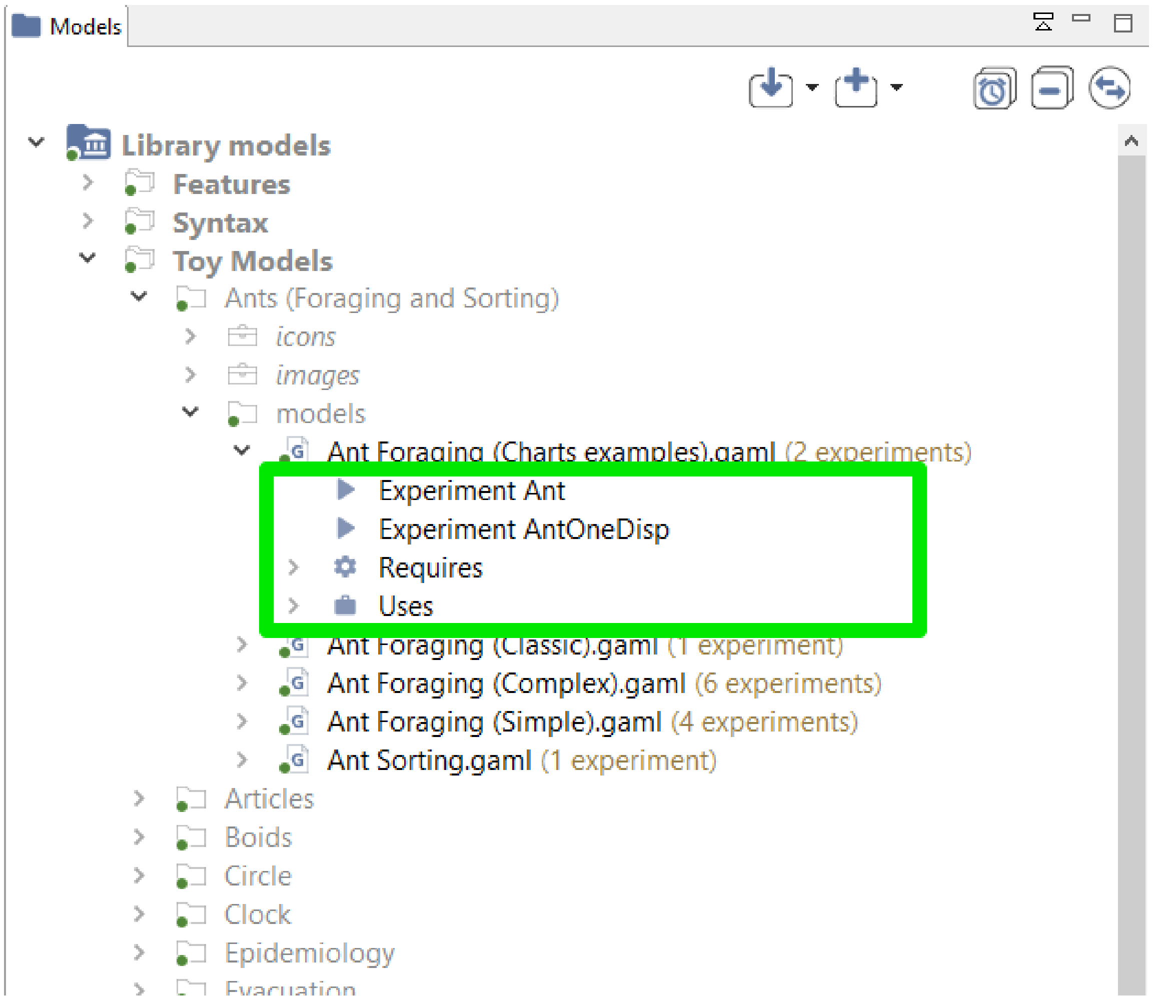
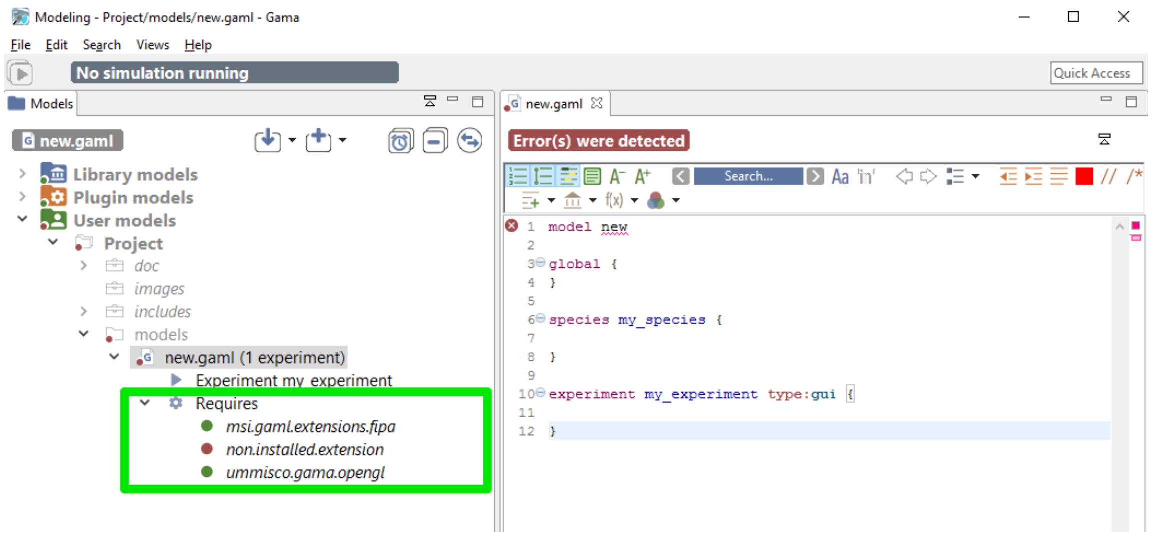

Models
Library models
Features
Syntax
Toy Models
Ants (Foraging and Sorting)
icons
images
models
Ant Foraging (Charts examples).gaml (2 experiments)
Experiment Ant
Experiment AntOneDisp
Requires
Uses
Ant Foraging (Classic).gaml (1 experiment)
Ant Foraging (Complex).gaml (6 experiments)
Ant Foraging (Simple).gaml (4 experiments)
Ant Sorting.gaml (1 experiment)
Articles
Boids
Circle
Clock
Epidemiology
Evacuation

Figure 7.7: images/inspect_model.png

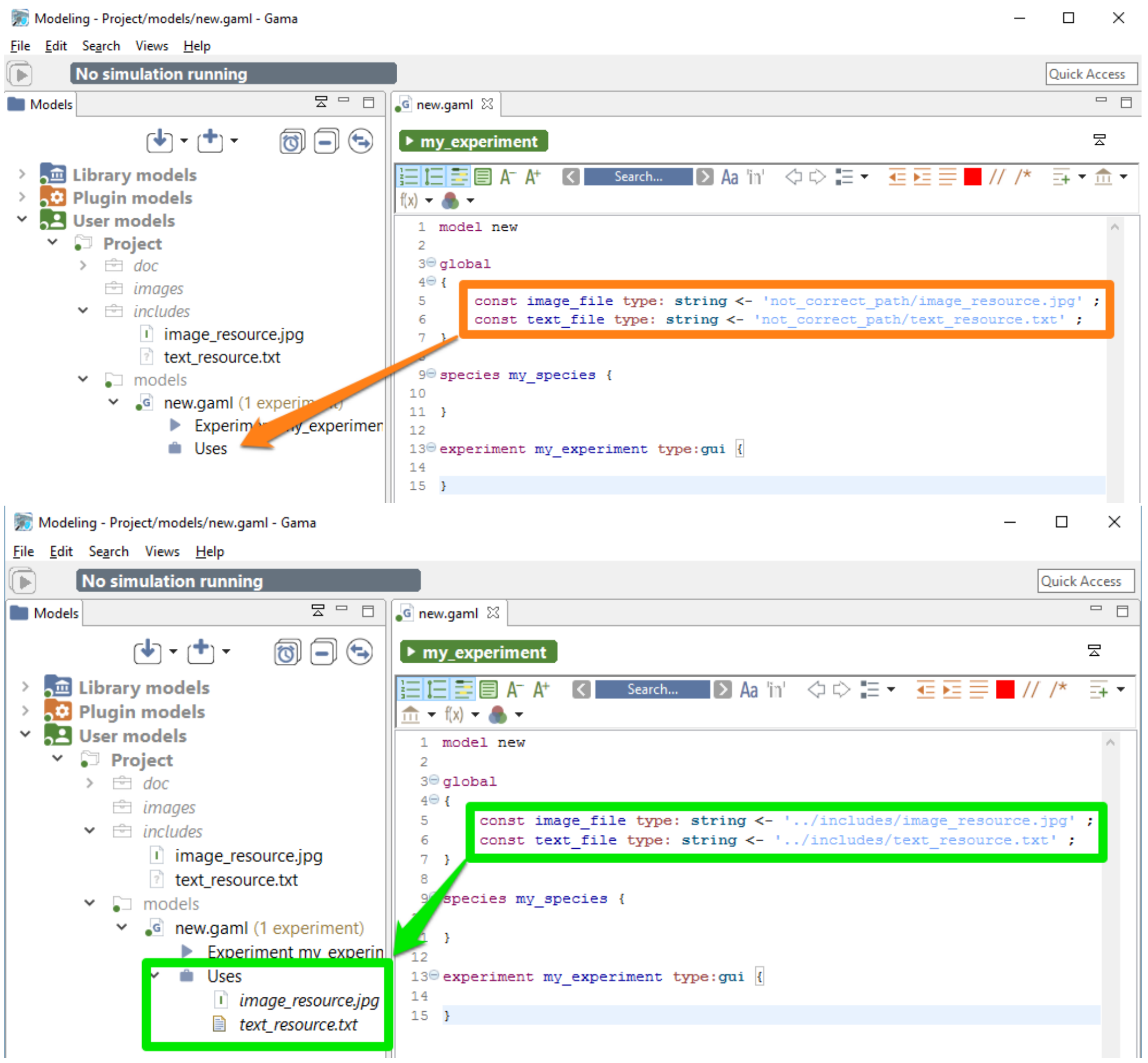
- **Require node** : The node *Require* is present if your model uses some plugins (additional or integrated by default). Each plugin is listed in this node, with a green icon if the plugin is already installed in your GAMA, and a red one if it is not the case. If the plugin you want in not installed, an error will be raised in your model. Please read about how to install plugins to learn some more about it.



```
model new

global {
}

species my_species {

}

experiment my_experiment type:gui {

}
```

Figure 7.8: images/requires_plugin_not_found.png

- **Uses node** : The node *Uses* is present if your model uses some external resources, *and if the path to the resource is correct* (if the path to the resource is not correct, the resource will not be displayed under *Uses*)



```
model new

global
{
	const image_file type: string <- 'not_correct_path/image_resource.jpg' ;
	const text_file type: string <- 'not_correct_path/text_resource.txt' ;
}

species my_species {

}

experiment my_experiment type:gui {

}
```

```
model new

global
{
	const image_file type: string <- '../includes/image_resource.jpg' ;
	const text_file type: string <- '../includes/text_resource.txt' ;
}

species my_species {

}

experiment my_experiment type:gui {

}
```

Moving Models Around

Model files, as well as resources, or even complete projects, can be moved around between the “Models Library”/“Plugin Models” and “Users Models” categories, or within them, directly in the *Navigator*. Drag’n drop operations are supported, as well as copy and paste. For example, the model “Life.gaml”, present in the “Models Library”, can perfectly be copied and then pasted in a project in the “Users Model”. This local copy in the workspace can then be further edited by the user without altering the original one.

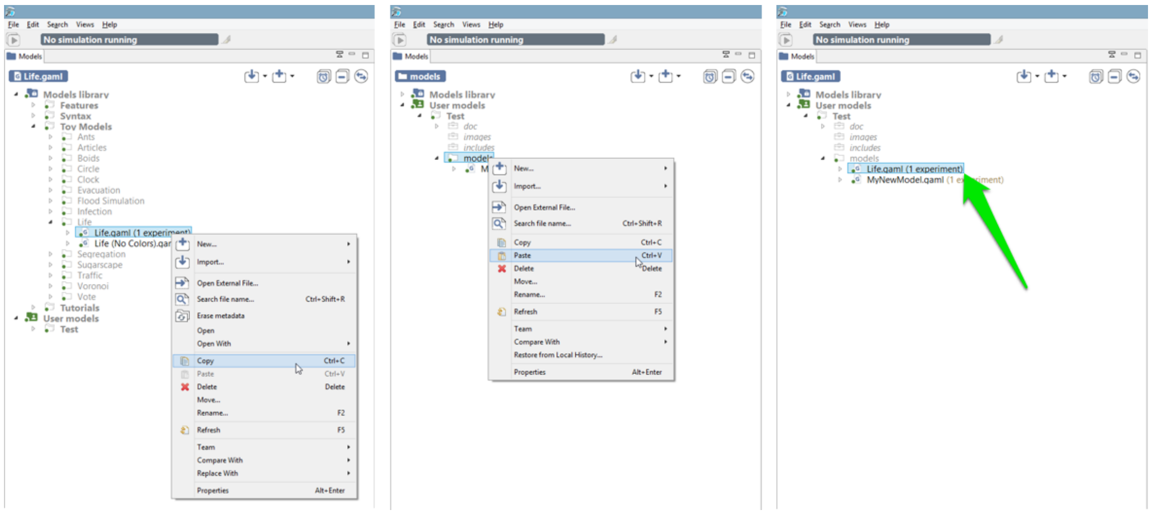
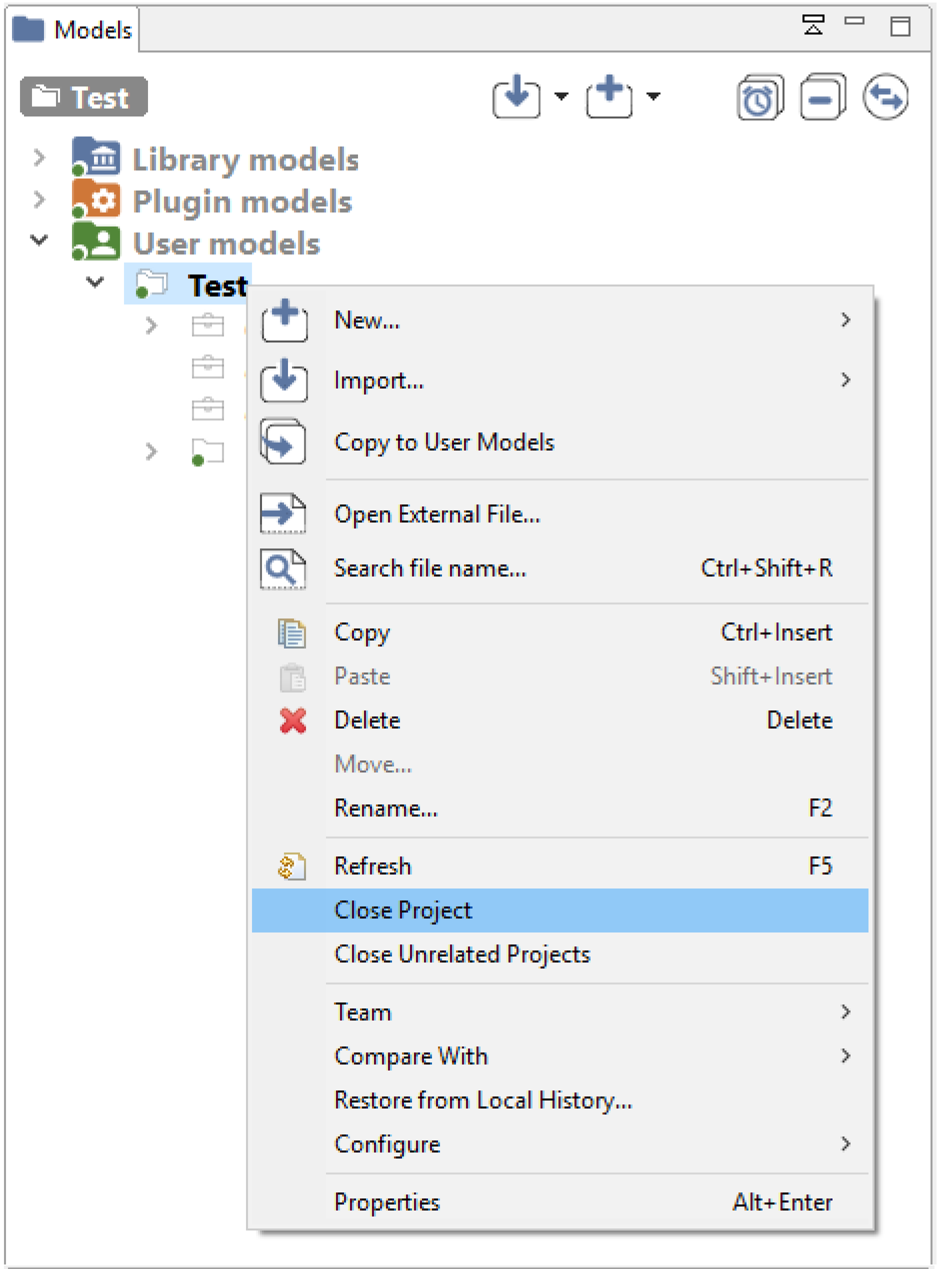
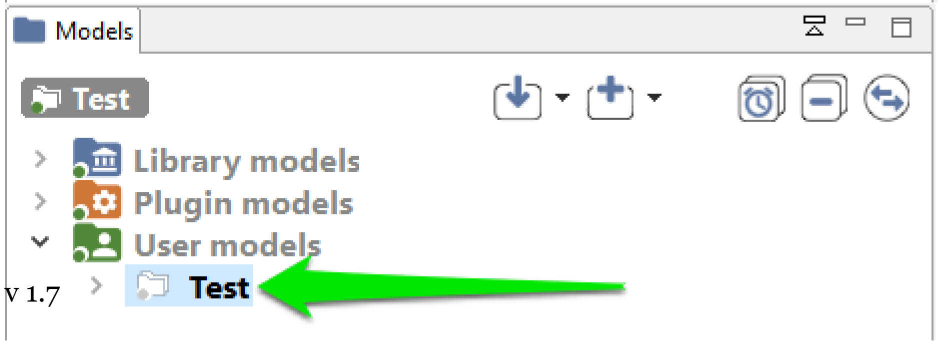


Figure 7.9: images/navigator_menu_copy_paste.png

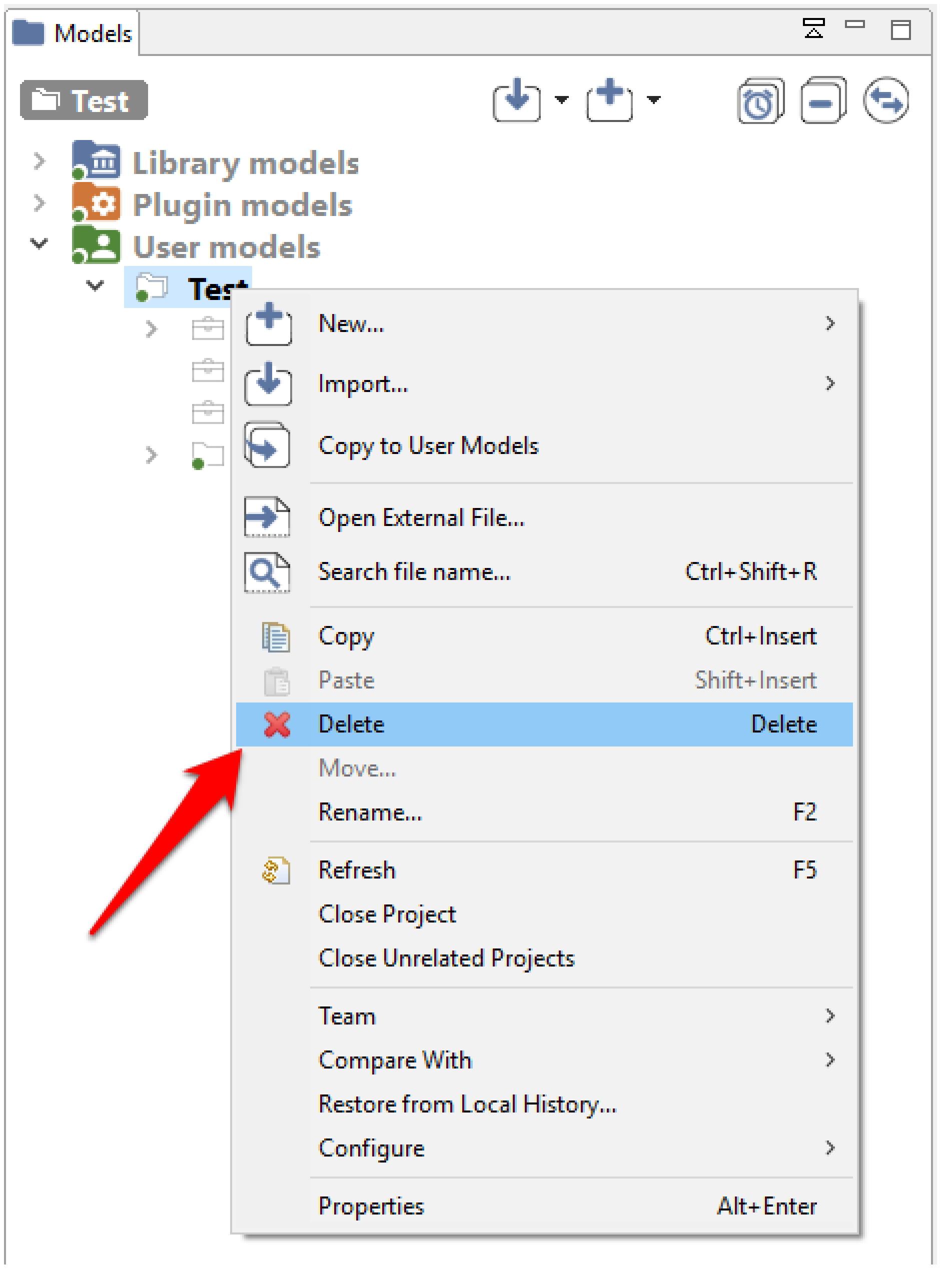
Closing and Deleting Projects

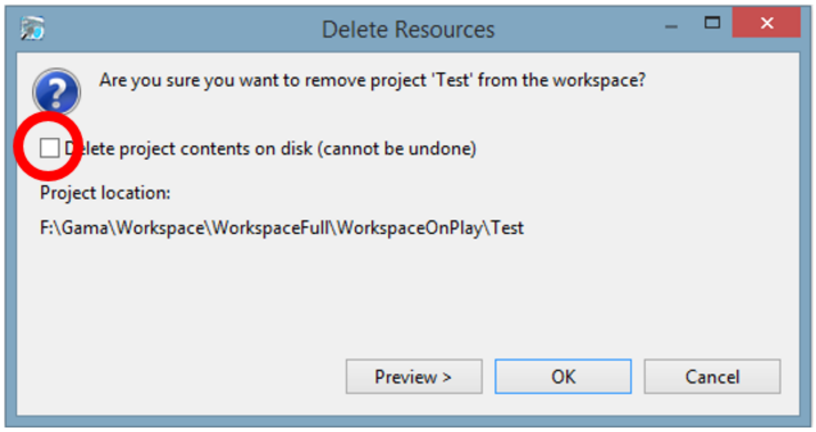
Users can choose to get rid of old projects by either **closing** or **deleting** them. Closing a project means that it will still reside in the workspace (and be still visible, although a bit differently, in the *Navigator*) but its model(s) won't participate to the build process and won't be displayable until the project is opened again.


Models
Test
Library models
Plugin models
User models
Test
New...
Import...
Copy to User Models
Open External File...
Search file name... Ctrl+Shift+R
Copy Ctrl+Insert
Paste Shift+Insert
Delete Delete
Move...
Rename... F2
Refresh F5
Close Project
Close Unrelated Projects
Team
Compare With
Restore from Local History...
Configure
Properties Alt+Enter


Models
Test
Library models
Plugin models
User models
Test

Deleting a project must be invoked when the user wants this project to not appear in the workspace anymore (unless, that is, it is imported again). Invoking this command will effectively make the workspace "forget" about this project, and this can be further doubled with a deletion of the projects resources and models from the filesystem.




Delete Resources

Are you sure you want to remove project 'Test' from the workspace?

Delete project contents on disk (cannot be undone)

Project location:

F:\Gama\Workspace\WorkspaceFull\WorkspaceOnPlay\Test

Chapter 8

Changing Workspace

It is possible, and actually common, to store different projects/models in different workspaces and to tell GAMA to switch between these workspaces. Doing so involves being able to create one or several new workspace locations (even if GAMA has been told to remember the current one) and being able to easily switch between them.

Table of contents

Switching to another Workspace

This process is similar to the choice of the workspace location when GAMA is launched for the first time. The only preliminary step is to invoke the appropriate command ("Switch Workspace") from the "File" menu.

In the dialog that appears, the current workspace location should already be entered. Changing it to a new location (or choosing one in the file selector invoked by clicking on "Browse…") and pressing "OK" will then either create a new workspace if none existed at that location or switch to this new workspace. Both operations will restart GAMA and set the new workspace

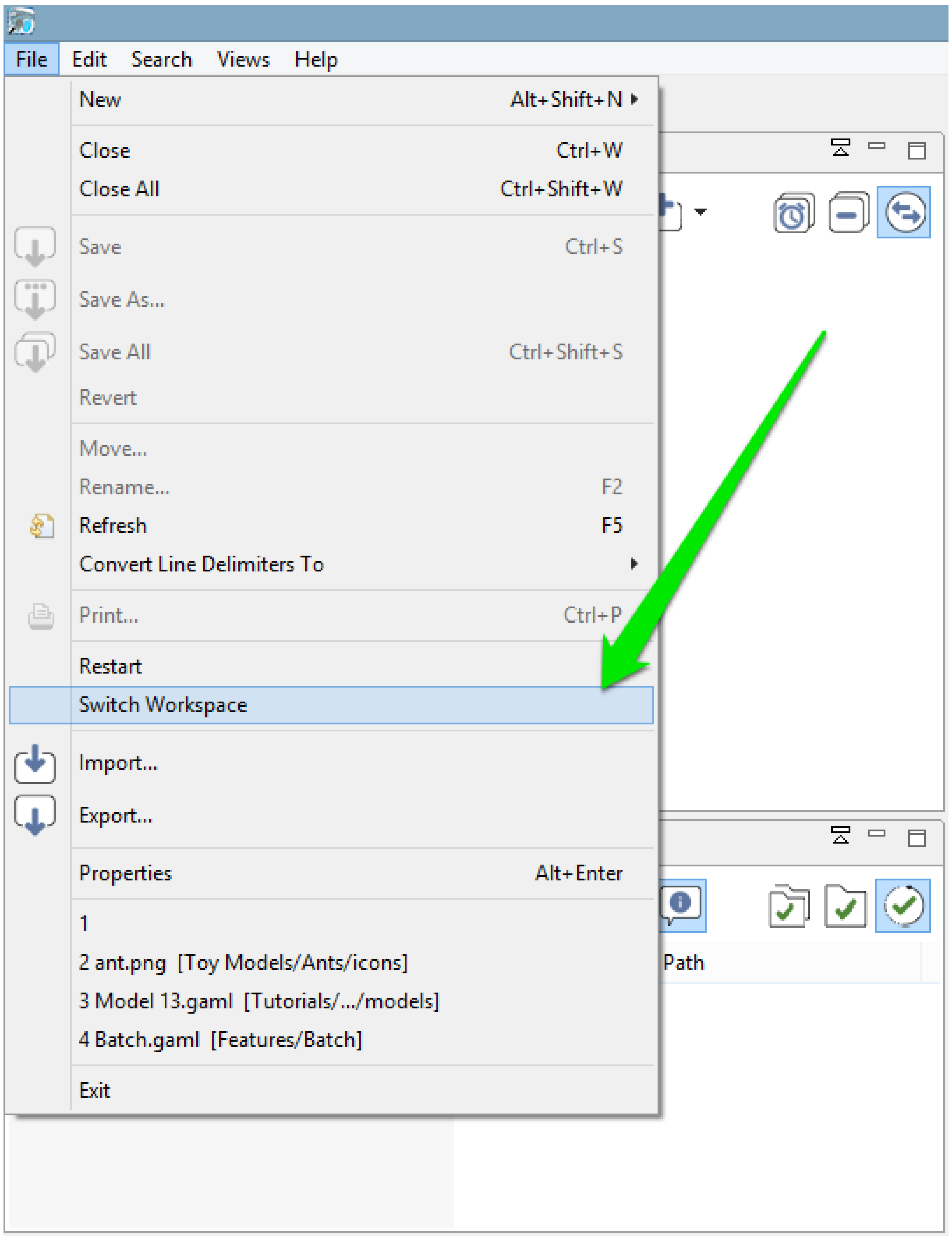
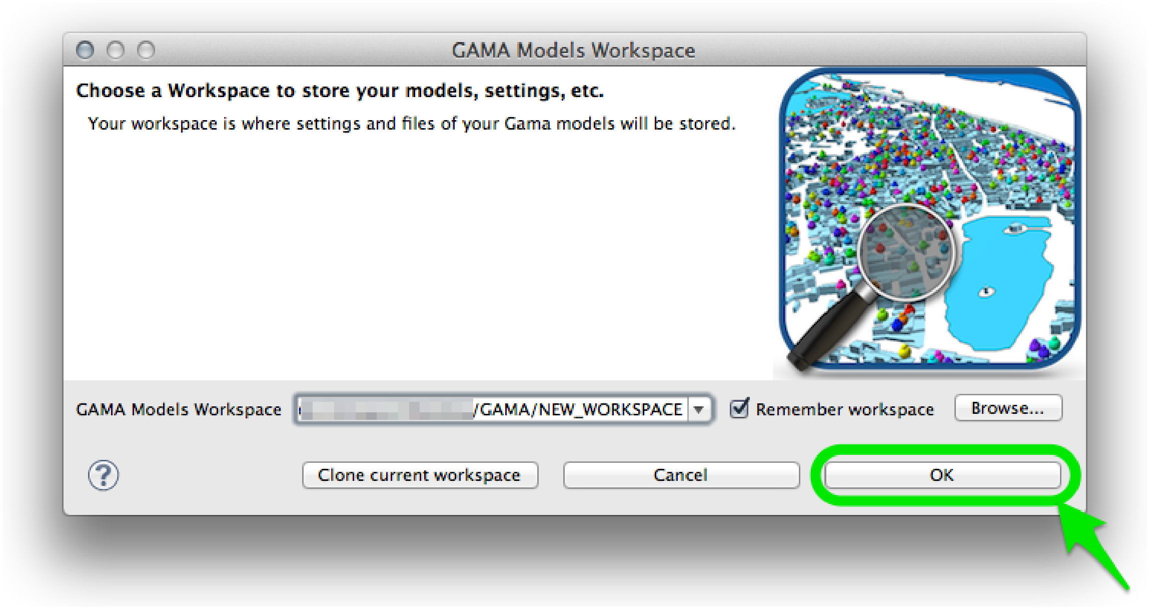


Figure 8.1: images/menu_switch.png

location. To come back to the previous location, just repeat this step (the previous location is normally now accessible from the combo box).


GAMA Models Workspace

Choose a Workspace to store your models, settings, etc.

Your workspace is where settings and files of your Gama models will be stored.

GAMA Models Workspace: /GAMA/NEW_WORKSPACE

☑ Remember workspace

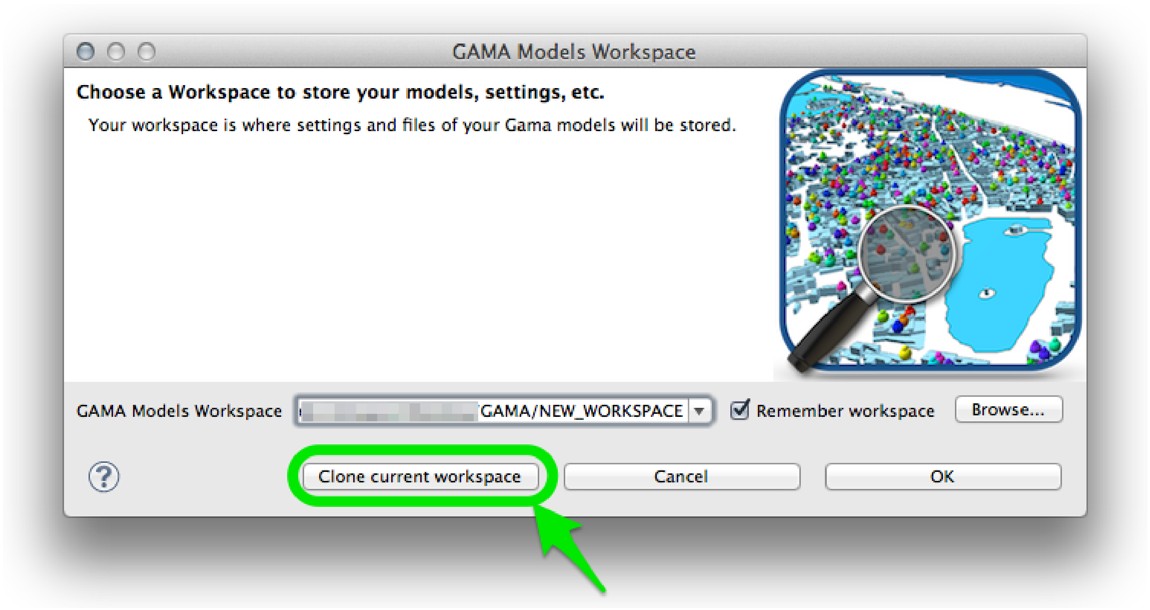
Figure 8.2: images/dialog_switch_ok.png

Cloning the Current Workspace

Another possibility, if you have models in your current workspace that you would like to keep in the new one (and that you do not want to import one by one after switching workspace), or if you change workspace because you suspect the current one is corrupted, or outdated, etc. but you still want to keep your models, is to **clone** the current workspace into a new (or existing) one.

Please note that cloning (as its name implies) is an operation that will make a *copy* of the files into a new workspace. So, if projects are stored in the current workspace, this will result in two different instances of the same projets/models with the same name in the two workspaces. However, for projects that are simply linked from the current workspace, only the link will be copied (which allows to have different workspaces "containing" the same project)

This can be done by entering the new workspace location and choosing "Clone current workspace" in the previous dialog instead of "Ok".


GAMA Models Workspace

Choose a Workspace to store your models, settings, etc.

Your workspace is where settings and files of your Gama models will be stored.

GAMA Models Workspace: GAMA/NEW_WORKSPACE

☑ Remember workspace

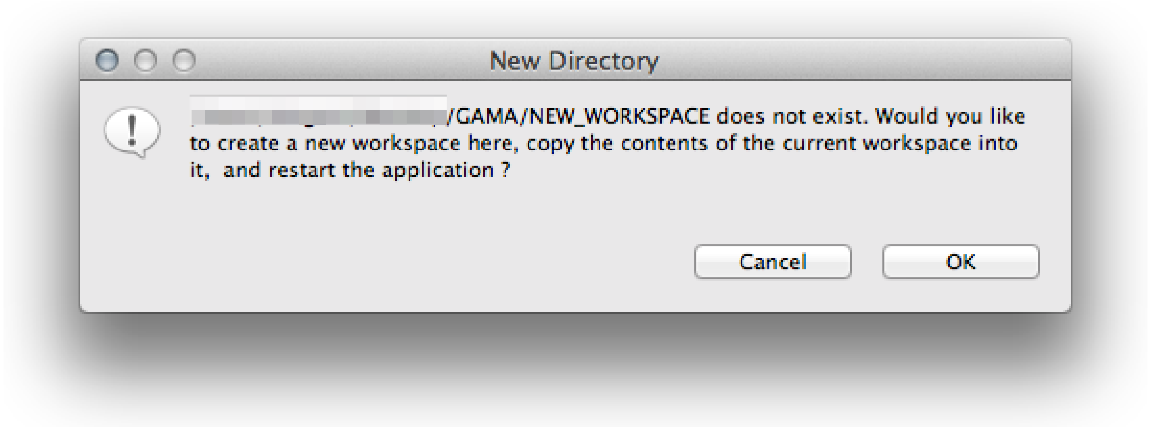
Figure 8.3: images/dialog_switch_clone.png

If the new location does not exist, GAMA will ask you to confirm the creation and cloning using a specific dialog box. Dismissing it will cancel the operation.

If the new location is already the location of an existing workspace, another confirmation dialog is produced. **It is important to note that all projects in the target workspace will be erased and replaced by the projects in the current workspace if you proceed**. Dismissing it will cancel the operation.

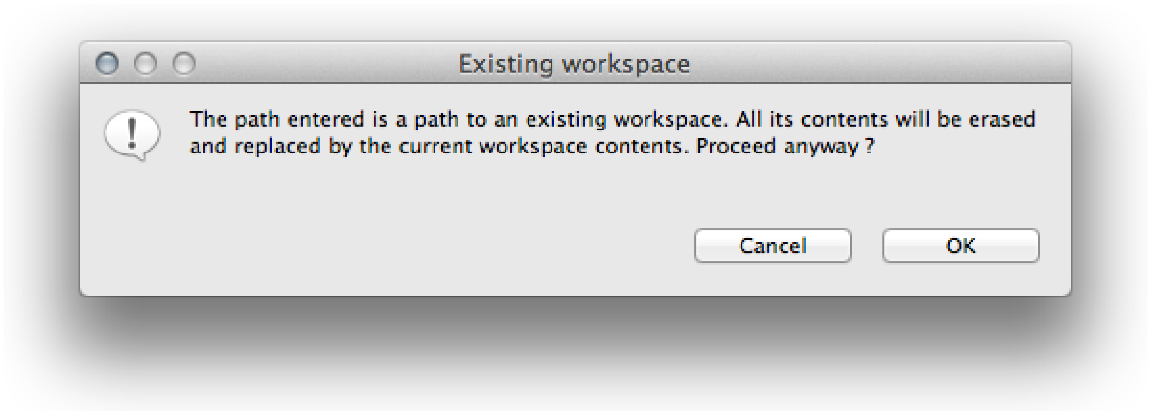
There are two cases where cloning is not accepted. The first one is when the user tries to clone the current workspace into itself (i.e. the new location is the same as the current location).

The second case is when the user tries to clone the current workspace into one of its subdirectories (which is not feasible).



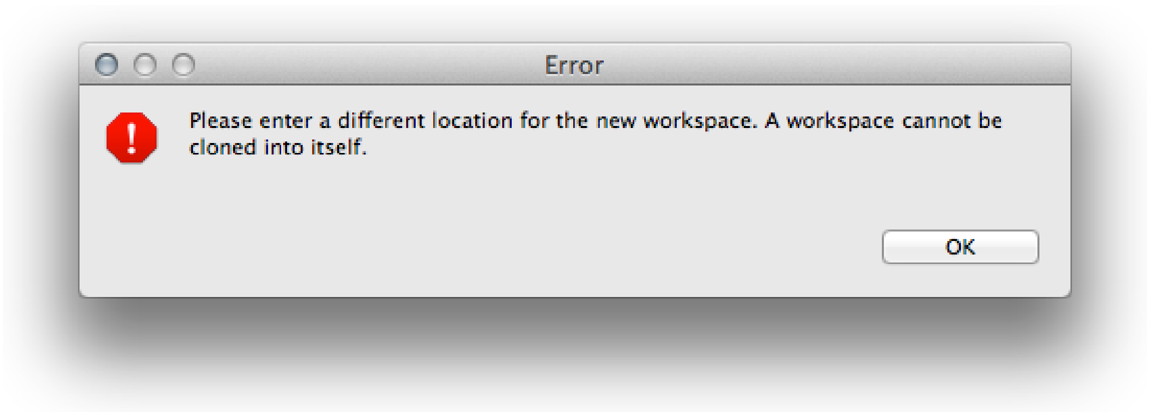
[redacted]/GAMA/NEW_WORKSPACE does not exist. Would you like to create a new workspace here, copy the contents of the current workspace into it, and restart the application ?

Figure 8.4: images/clone_confirm_new.png



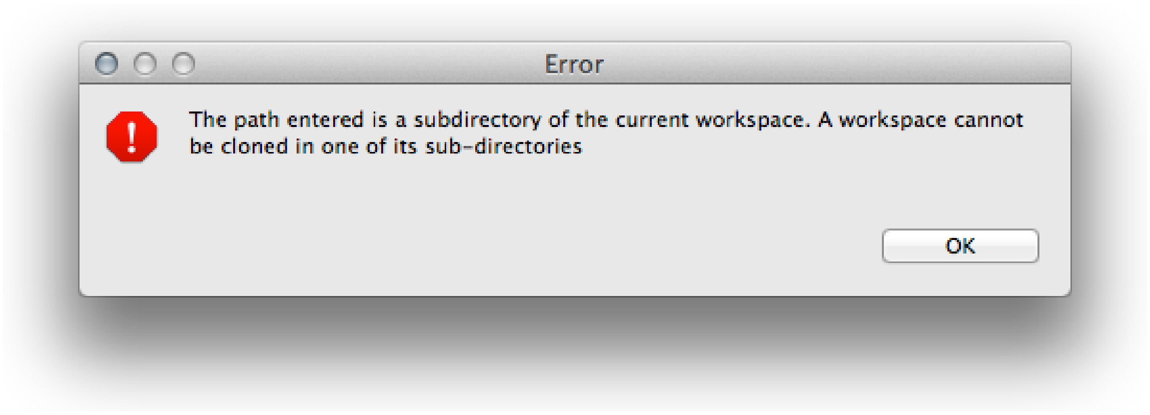
The path entered is a path to an existing workspace. All its contents will be erased and replaced by the current workspace contents. Proceed anyway ?

Figure 8.5: images/clone_confirm_existing.png



Please enter a different location for the new workspace. A workspace cannot be cloned into itself.

Figure 8.6: images/close_error_same.png



The path entered is a subdirectory of the current workspace. A workspace cannot be cloned in one of its sub-directories

Figure 8.7: images/close_error_subdir.png

Chapter 9

Importing Models

Importing a model refers to making a model file (or a complete project) available for edition and experimentation in the **workspace**. With the exception of headless experiments, GAMA requires that models be manageable in the current workspace to be able to validate them and eventually experiment them.

There are many situations where a model needs to be *imported* by the user: someone sent it to him/her by mail, it has been attached to an issue report, it has been shared on the web or an SVN server, or it belongs to a previous workspace after the user has switched workspace. The instructions below apply equally to all these situations.

Since model files need to reside in a project to be managed by GAMA, it is usually preferable to import a whole project rather than individual files (unless, of course, the corresponding models are simple enough to not require any additional resources, in which case, the model file can be imported with no harm into an existing project). GAMA will then try to detect situations where a model file is imported alone and, if a corresponding project can be found (for instance, in the upper directories of this file), to import the project instead of the file. As the last resort, GAMA will import orphan model files into a *generic* project called *"Unclassified Models"* (which will be created if it does not exist yet).

Table of contents

– Drag'n Drop / Copy-Paste Limitations

The "Import…" Menu Command

The simplest, safest and most secure way to import a project into the workspace is to use the built-in "Import…" menu command, available in the "File" menu or in the contextual menu of the *Navigator*.

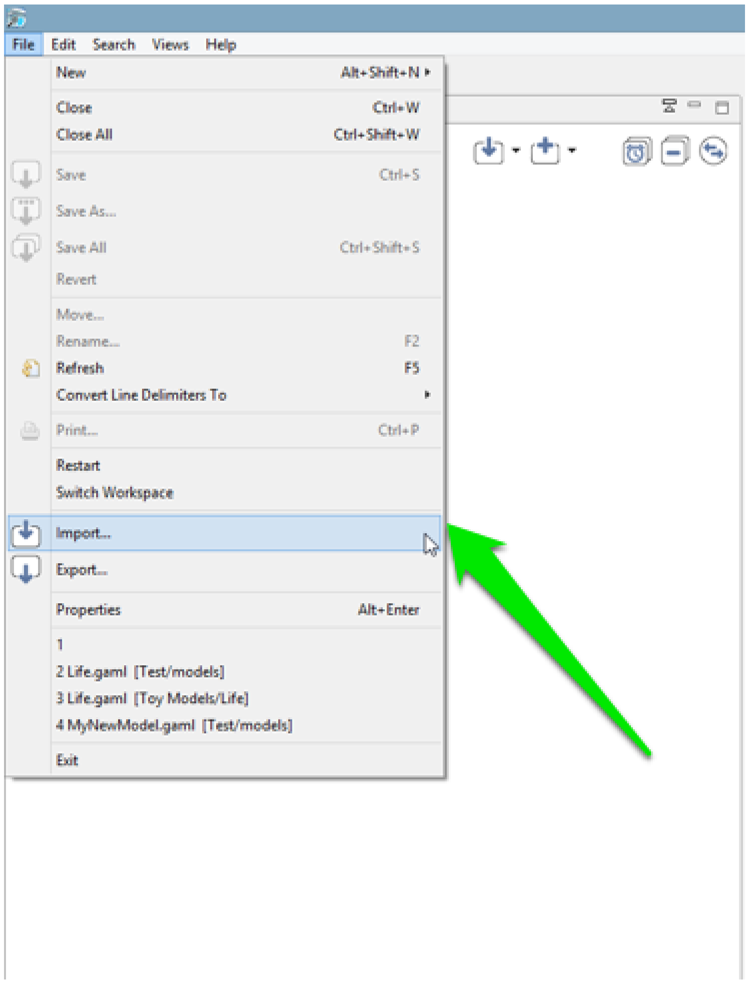


Figure 9.1: images/menu_file_import.png

When invoked, this command will open a dialog asking the user to choose the source of the importation. It can be a directory in the filesystem (in which GAMA will look for existing projects), a zip file, a SVN site, etc. It is safer in any case to choose "Existing Projects into Workspace".

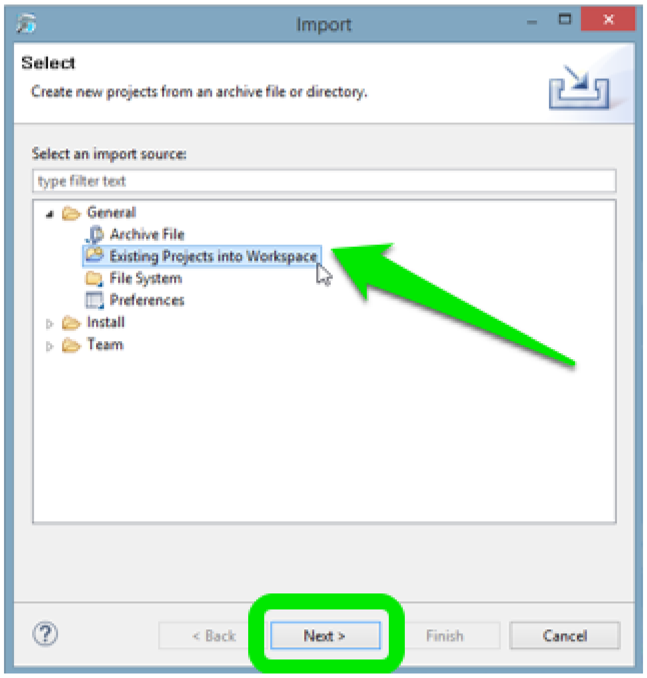


Figure 9.2: images/dialog_import.png

Note that when invoked from the contextual menu, "Import…" will directly give access to a shortcut of this source in a submenu.

Both options will lead the user to a last dialog where he/she will be asked to:

1. Enter a location (or browse to a location) containing the GAMA project(s) to import
2. Choose among the list of available projects (computed by GAMA) the ones to effectively import
3. Indicate whether or not these projects need to be **copied to** or **linked from** the workspace (the latter is done by default)

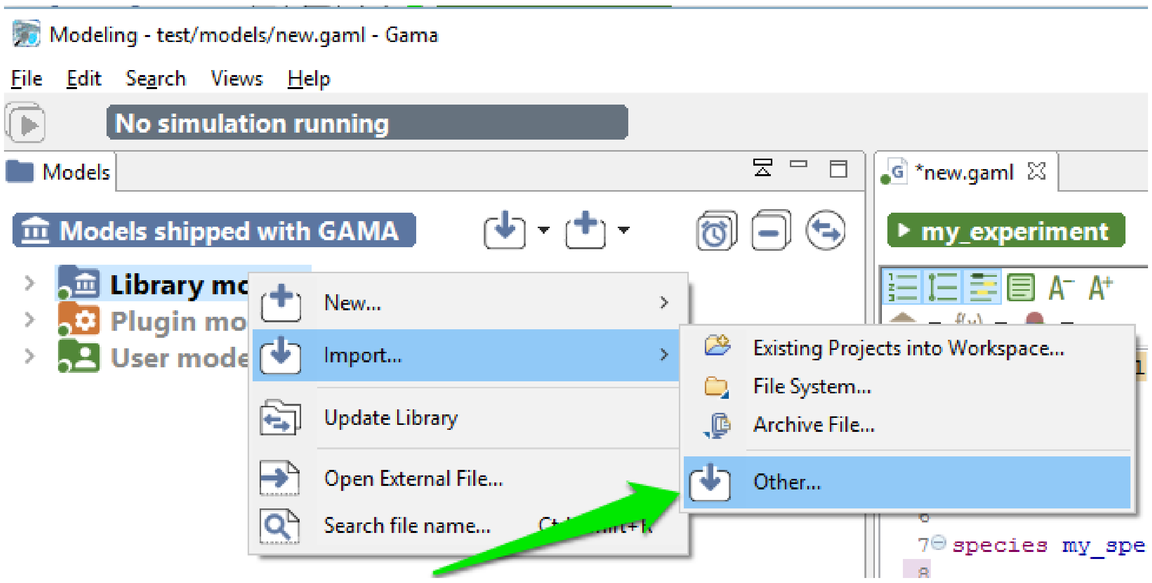


Figure 9.3: images/menu_navigator_import.png

Silent import

Another (possibly simpler, but less controllable) way of importing projects and models is to either pass a path to a model when launching GAMA from the command line or to double-click on a model file (ending in *.gaml*) in the Explorer or Finder (depending on your OS).

If the file is not already part of an imported project in the current workspace, GAMA will:

1. silently import the project (by creating a link to it),
2. open an editor on the file selected.

This procedure may fail, however, if a project of the same name (but in a different location) already exists in the workspace, in which case GAMA will refuse to import the project (and hence, the file). The solution in this case is to rename the project to import (or to rename the existing project in the workspace).

Drag'n Drop / Copy-Paste Limitations

Currently, **there is no way** to drag and drop an entire project into GAMA *Navigator* (or to copy a project in the filesystem and paste it in the *Navigator*). Only individual model

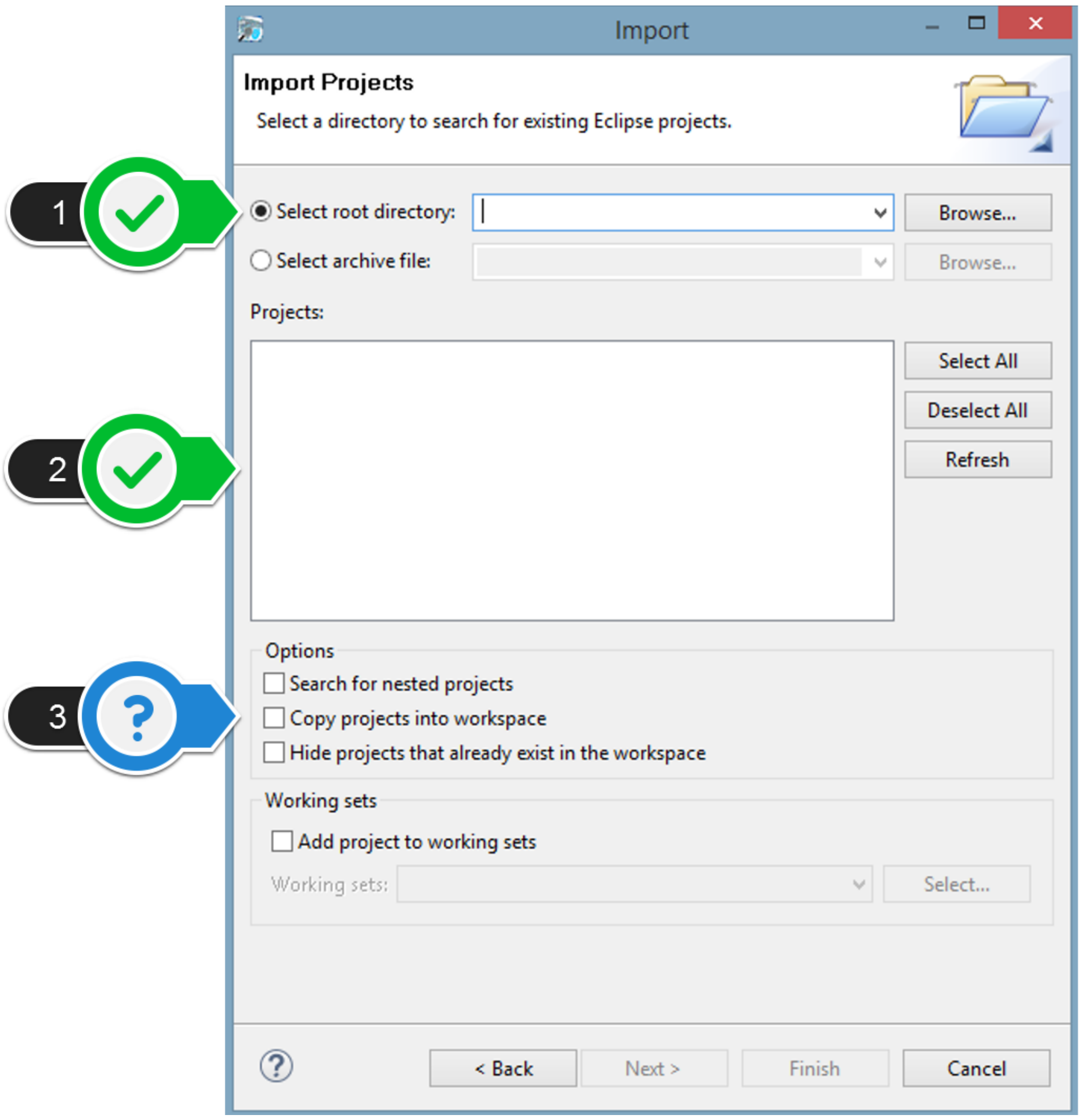

Import
Import Projects
Select a directory to search for existing Eclipse projects.
1
Select root directory:
Browse...
Select archive file:
Browse...
Projects:
Select All
Deselect All
2
Refresh
Options
3
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel

Figure 9.4: images/dialog_import_2.png

files, folders or resources can be moved this way (and they have to be dropped or pasted into existing projects).

This limitation might be removed some time in the future, however, allowing users to use the *Navigator* as a project drop or paste target, but it is not the case yet.

Editing Models

Editing models in GAMA is very similar to editing programs in a modern IDE like Eclipse. After having successfully launched the program, the user has two fundamental concepts at its disposal: a **workspace**, which contains models or links to models organized like a hierarchy of files in a filesystem, and the **workbench** (aka, the *main window*), which contains the tools to create, modify and experiment these models.

Understanding how to navigate in the **workspace** is covered in another section and, for the purpose of this section, we just need to understand that it is organized in **projects**, which contain **models** and their associated data. **Projects** are further categorized, in GAMA, into three categories : *Models Library* (built-in models shipped with GAMA and automatically linked from the workspace), *Shared Models*, and *User Models*.

This section covers the following sub-sections :

- 1. GAML Editor Generalities
- 2. GAML Editor Toolbar
- 3. Validation of Models
- 4. Graphical Editor

Chapter 10

The GAML Editor - Generalities

The GAML Editor is a text editor that proposes several services to support the modeler in writing correct models: an integrated live validation system, a ribbon header that gives access to experiments, information, warning and error markers.

Table of contents

Creating a first model

Editing a model requires that at least one **project** is created in *User Models*. If there is none, right-click on *User Models* and choose "New… > Gama Project…" (if you already have user projects and want to create a model in one of them, skip the next step).

A dialog is then displayed, offering you to enter the name of the project as well as its location on the filesystem. Unless you are absolutely sure of what you are doing, keep the "Use default location" option checked. An error will be displayed if the project name already exists in the

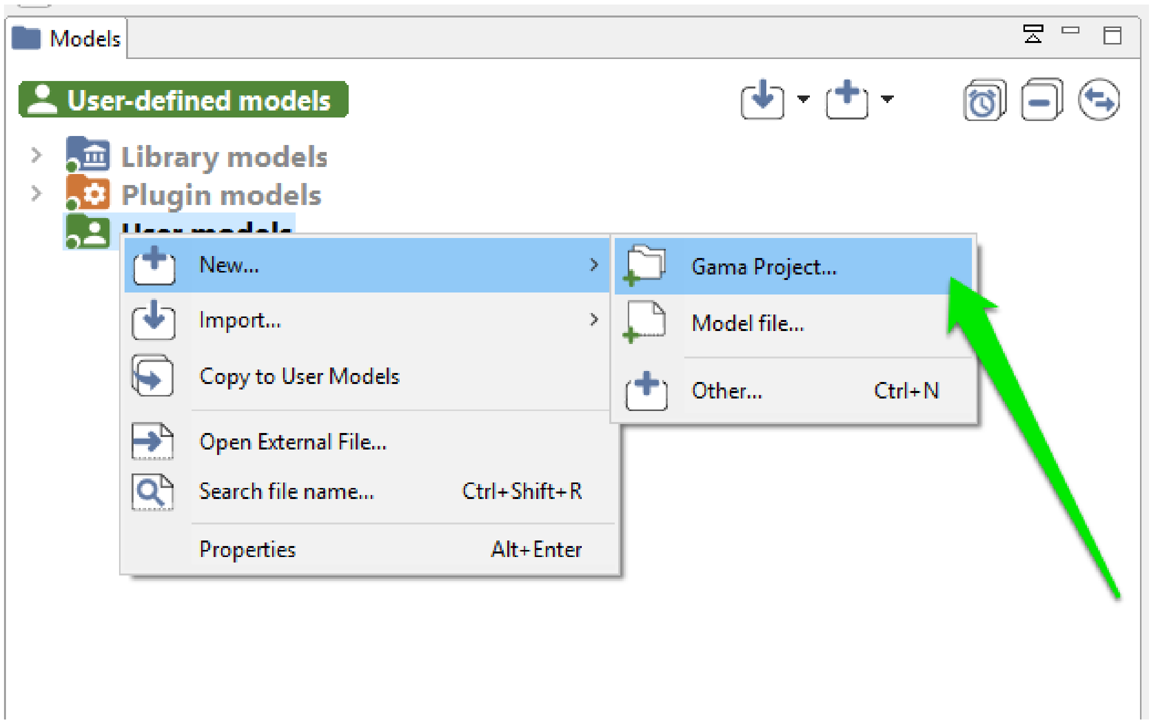
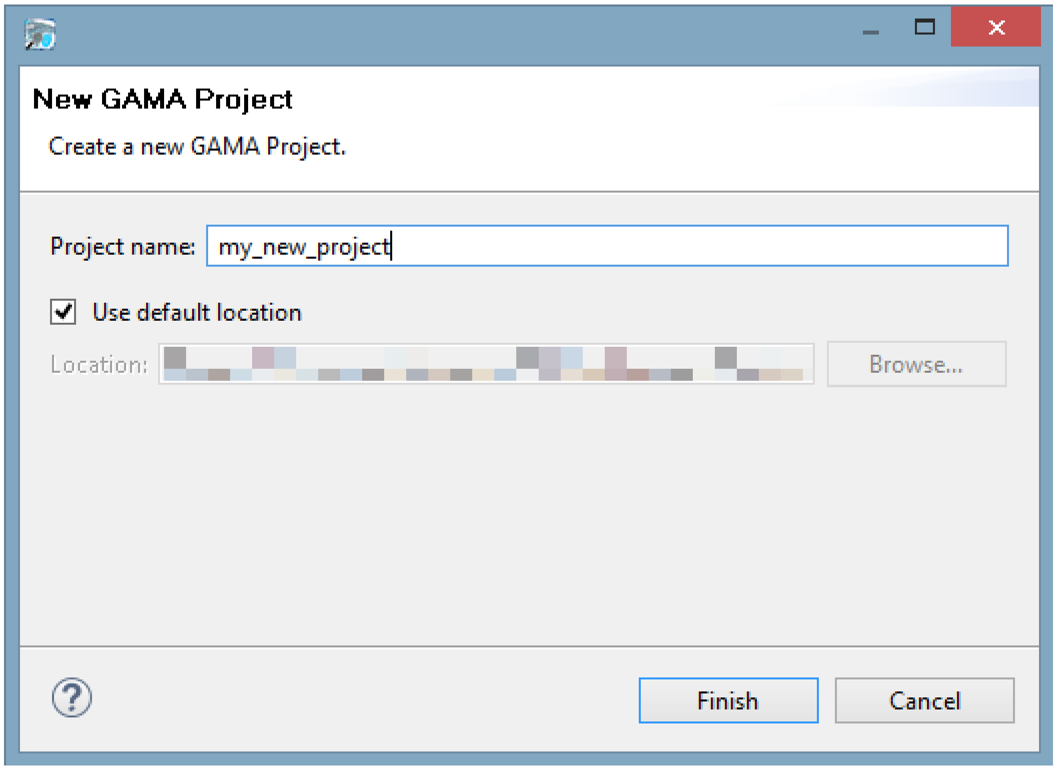


Figure 10.1: images/1.new_project.png

workspace, in which case you will have to change it. Two projects with similar names can not coexist in the workspace (even if they belong to different categories).


New GAMA Project

Create a new GAMA Project.

Project name: my_new_project

☑ Use default location

Location:

Figure 10.2: images/2.new_project2.png

Once the project is created (or if you have an existing project), navigate to it and right-click on it. This time, choose “New…>Model file…” to create a new model.

A new dialog is displayed, which asks for several required or optional information. The *Container* is normally the name of the project you have selected, but you can choose to place the file elsewhere. An error will be displayed if the container does not exist (yet) in the workspace. You can then choose whether you want to use a template or not for producing the initial file, and you are invited to give this file a name. An error is displayed if this name already exists in this project. The name of the model, which is by default computed with respect to the name of the file, can be actually completely different (but it may not contain white spaces or punctuation characters). The name of the author, as well as the textual description of the model and the creation of an HTML documentation, are optional.

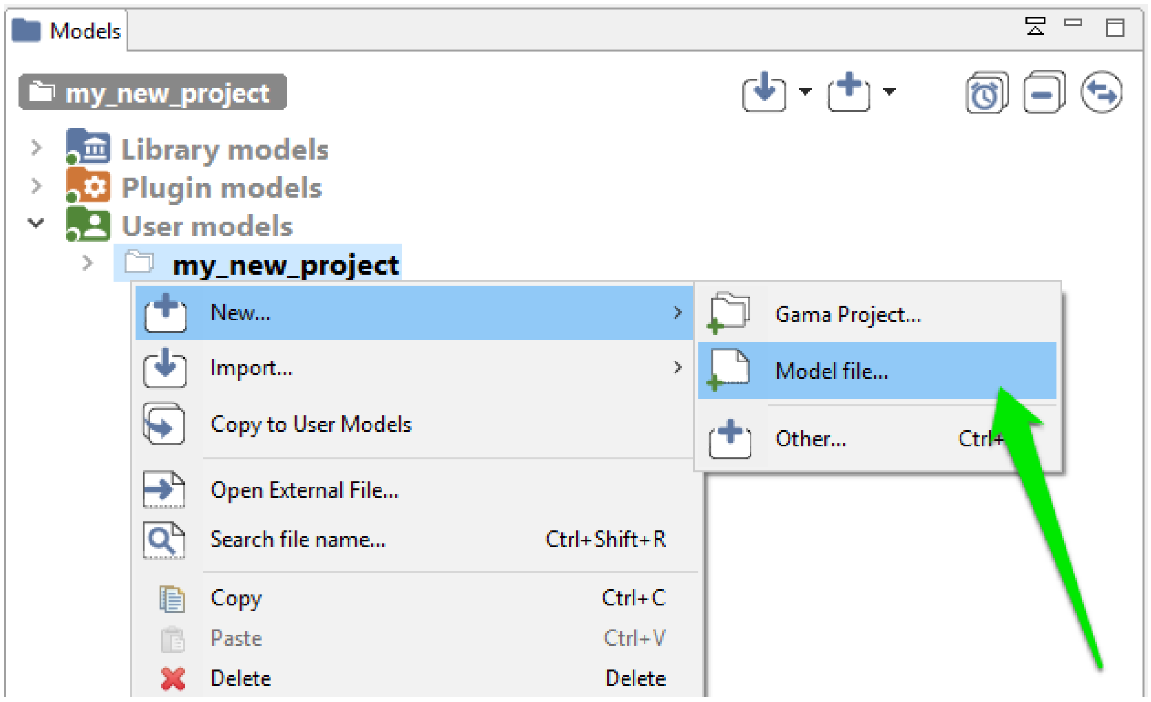
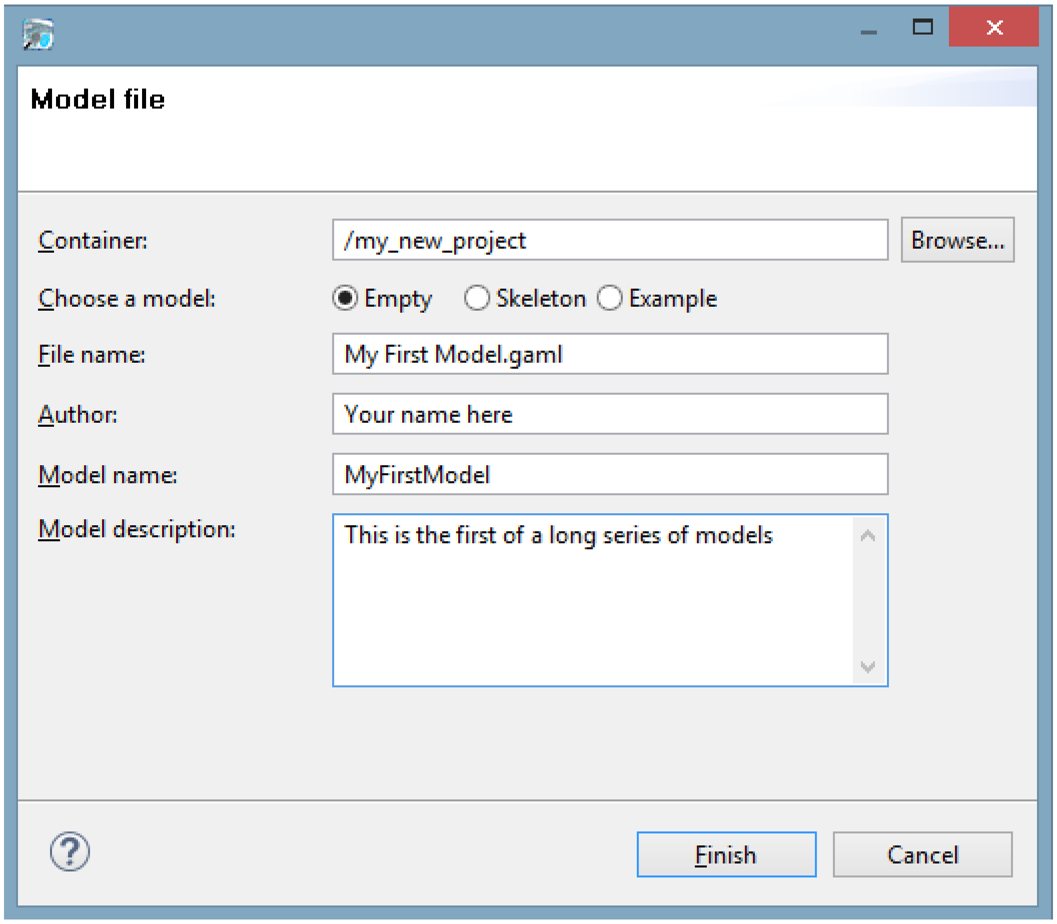


Figure 10.3: images/3.new_model.png



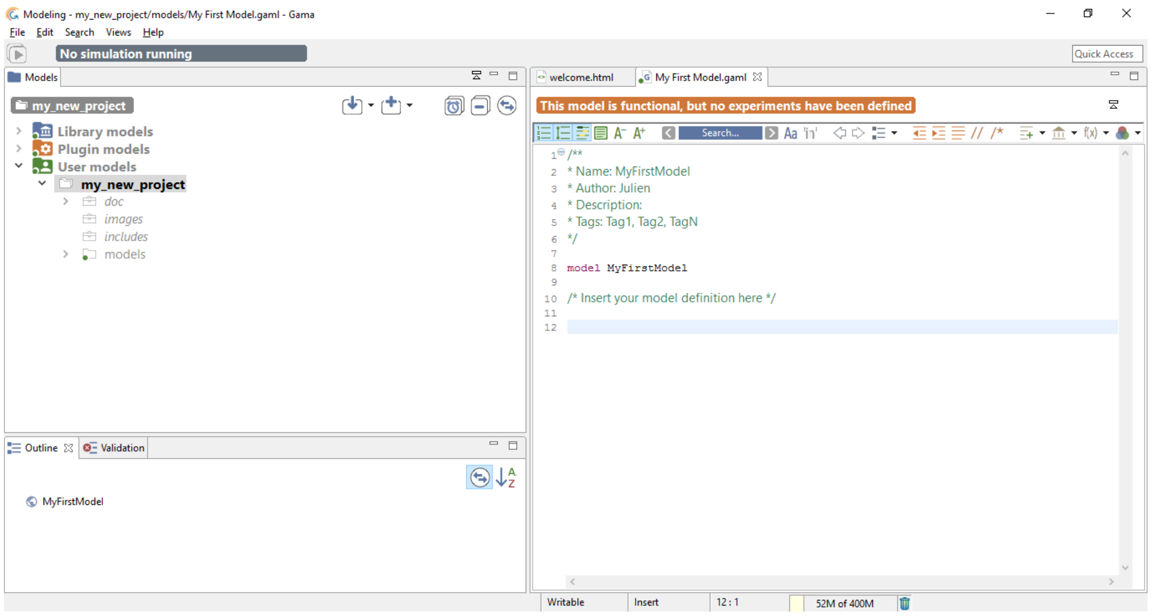
Model file

Container:	/my_new_project
Choose a model:	[● Empty] ○ Skeleton ○ Example
File name:	My First Model.gaml
Author:	Your name here
Model name:	MyFirstModel
Model description:	This is the first of a long series of models

Figure 10.4: images/4.new_model2.png

Status of models in editors

Once this dialog is filled and accepted, GAMA will display the new "empty" model.



This model is functional, but no experiments have been defined

```
/**
 * Name: MyFirstModel
 * Author: Julien
 * Description:
 * Tags: Tag1, Tag2, TagN
 */

model MyFirstModel

/* Insert your model definition here */

```

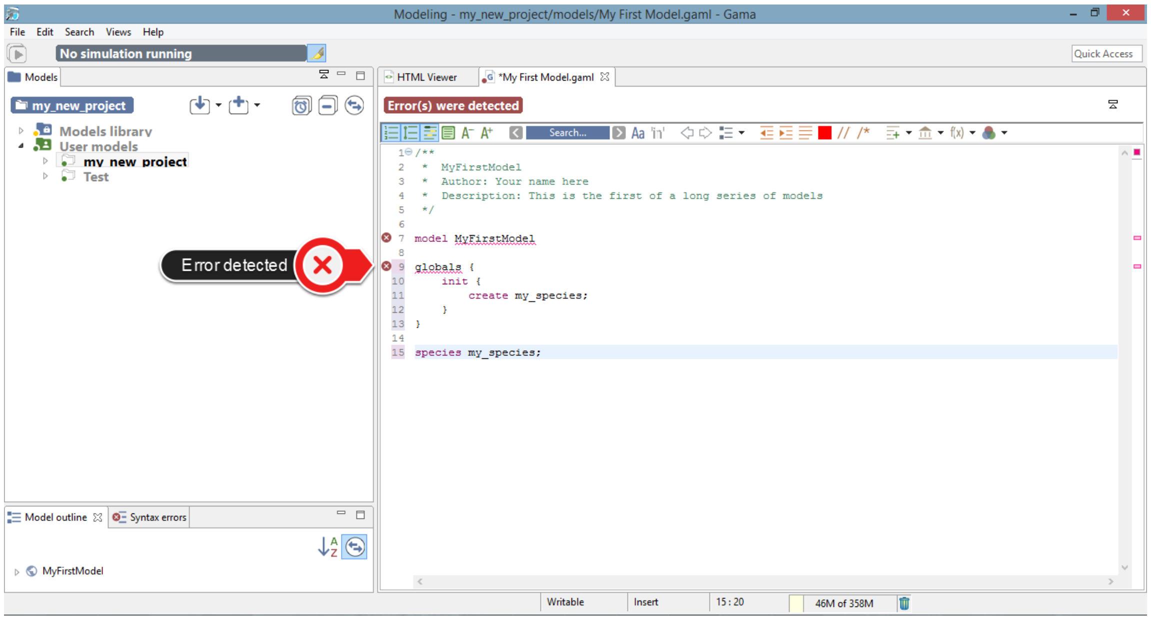
Figure 10.5: images/5.view_model.png

Although GAML files are just plain text files, and can therefore be produced or modified in any text processor, using the dedicated GAML editor offers a number of advantages, among which the live display of errors and model statuses. A model can actually be in four different states, which are visually accessible above the editing area: *Functional* (orange color), *Experimentable* (green color), *InError* (red color), InImportedError_(yellow color). See the section on model compilation for more precise information about these statuses._

In its initial state, a model is always in the *Functional* state, which means it compiles without problems, but cannot be used to launch experiments. The *InError* state, depicted below, occurs when the file contains errors (syntactic or semantic ones).

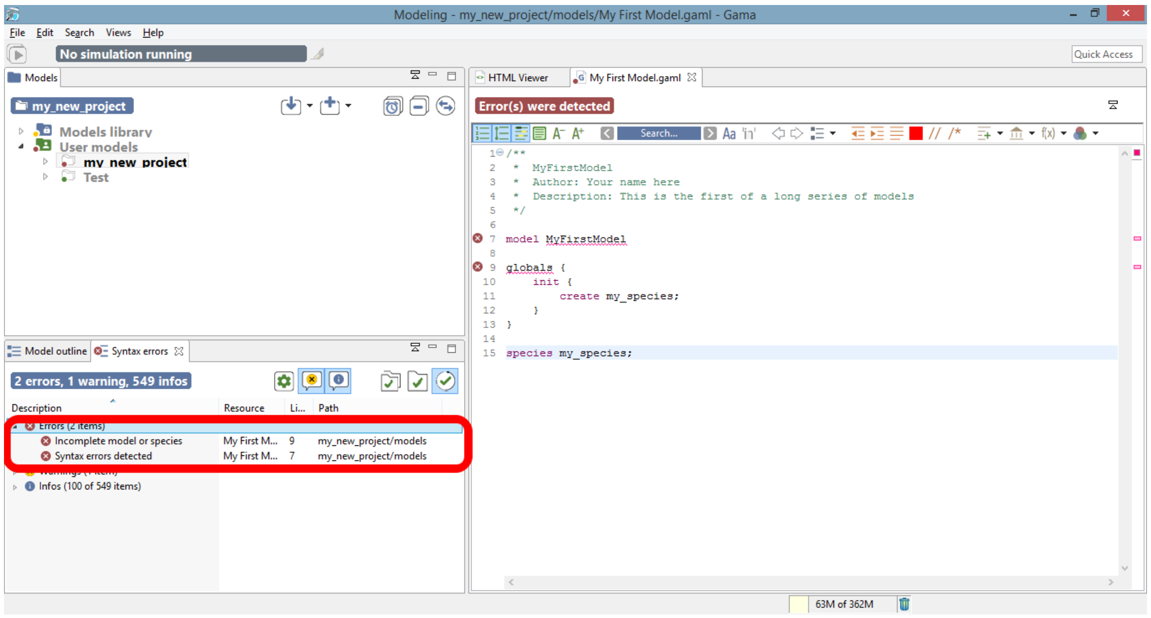
While the file is not saved, these errors remain displayed in the editor and nowhere else. If you save the file, they are now considered as "workspace errors" and get displayed in the "Problems" view below the editor.

Reaching the *Experimentable* state requires that all errors are eliminated and that at least one experiment is defined in the model, which is the case now in our toy model. The experiment is immediately displayed as a button in the toolbar, and clicking on it will allow to



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */

model MyFirstModel

globals {
    init {
        create my_species;
    }
}

species my_species;
```

Figure 10.6: images/6.view_model_with_error.png

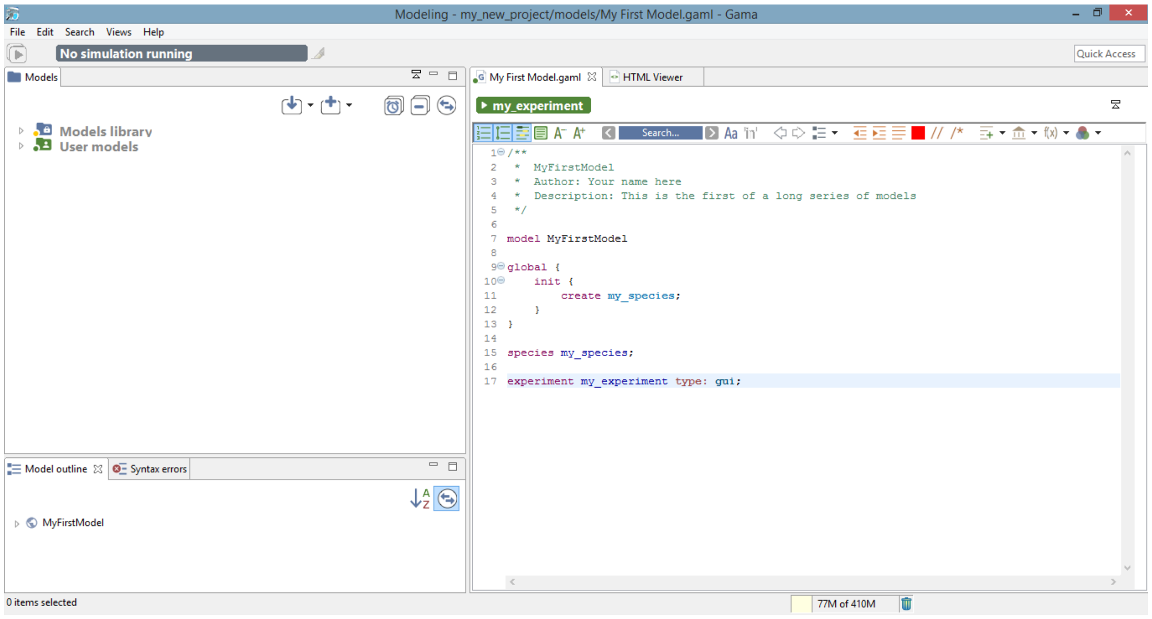


```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */

model MyFirstModel

globals {
    init {
        create my_species;
    }
}

species my_species;
```

Description	Resource	Li...	Path
Errors (2 items)			
Incomplete model or species	My First M...	9	my_new_project/models
Syntax errors detected	My First M...	7	my_new_project/models
Warnings (1 item)			
Infos (100 of 549 items)			

Figure 10.7: images/7.view_model_with_error_saved.png

launch this experiment on your model. See the section about running experiments for more information on this point.



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */

model MyFirstModel

global {
    init {
        create my_species;
    }
}

species my_species;

experiment my_experiment type: gui;
```

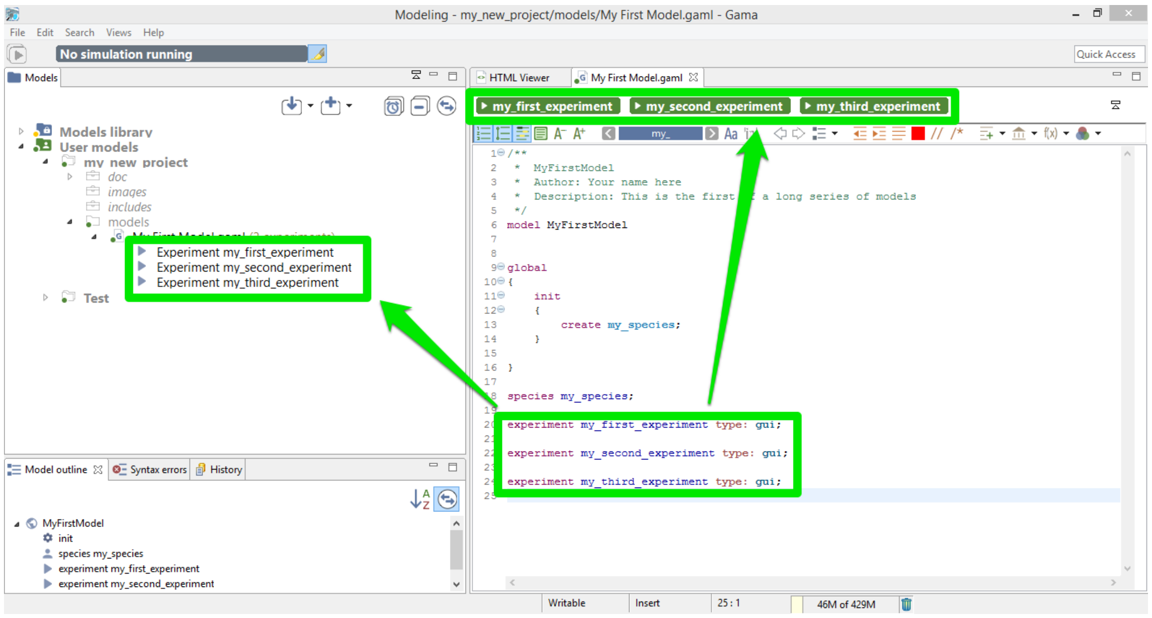
Figure 10.8: images/8.view_model_with_experiment.png

Experiment buttons are updated in real-time to reflect what's in your code. If more than one experiment is defined, corresponding buttons will be displayed in addition to the first one.

Editor Preferences

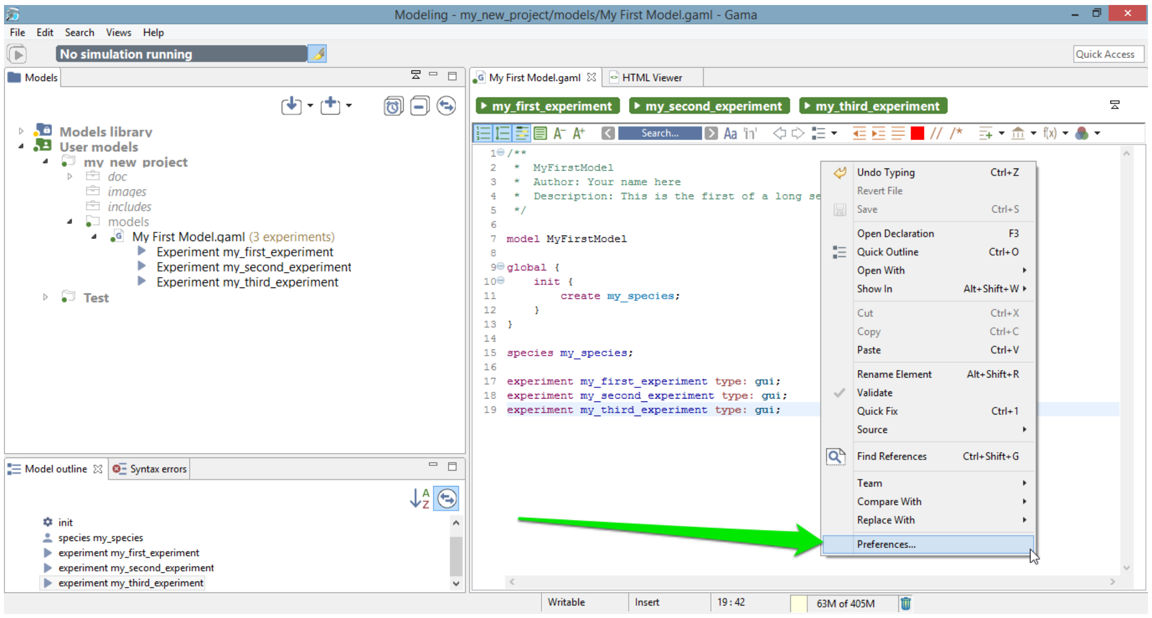
Text editing in general, and especially in Eclipse-based editors, sports a number of options and preferences. You might want to turn off/on the numbering of the lines, change the fonts used, change the colors used to highlight the code, etc. All of these preferences are accessible from the "Preferences…" item of the editor contextual menu.

Explore the different items present there, keeping in mind that these preferences will apply to all the editors of GAMA and will be stored in your workspace.



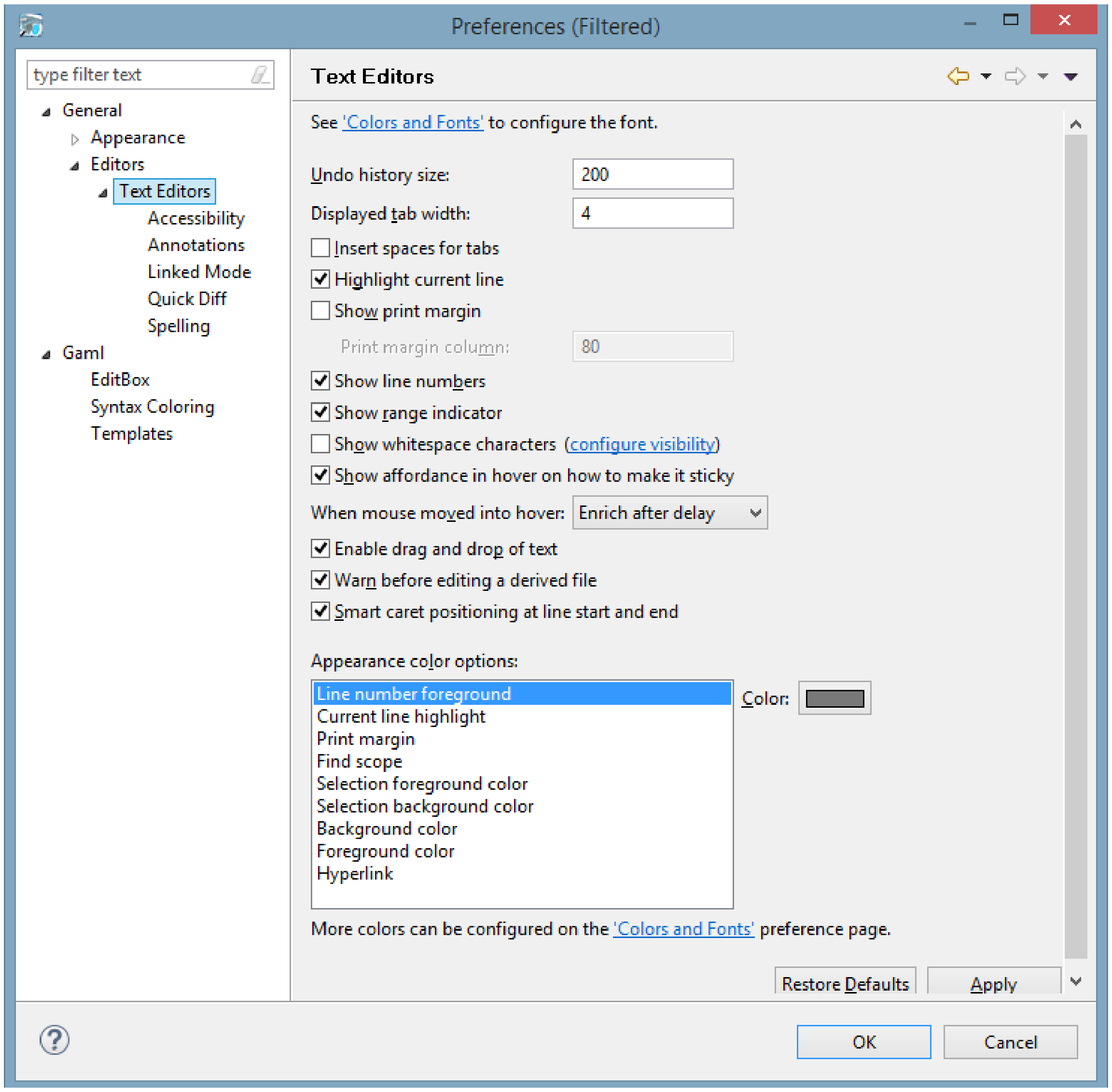
```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        create my_species;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 10.9: images/9.view_model_with_3_experiments.png



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long se
 */

model MyFirstModel

global {
    init {
        create my_species;
    }
}

species my_species;

experiment my_first_experiment type: gui;
experiment my_second_experiment type: gui;
experiment my_third_experiment type: gui;
```

Figure 10.10: images/10.view_model_with_preferences.png



Text Editors

See 'Colors and Fonts' to configure the font.

Undo history size: 200

Displayed tab width: 4

☐ Insert spaces for tabs

☑ Highlight current line

☐ Show print margin

Print margin column: 80

☑ Show line numbers

☑ Show range indicator

☐ Show whitespace characters (configure visibility)

☑ Show affordance in hover on how to make it sticky

When mouse moved into hover: Enrich after delay

☑ Enable drag and drop of text

☑ Warn before editing a derived file

☑ Smart caret positioning at line start and end

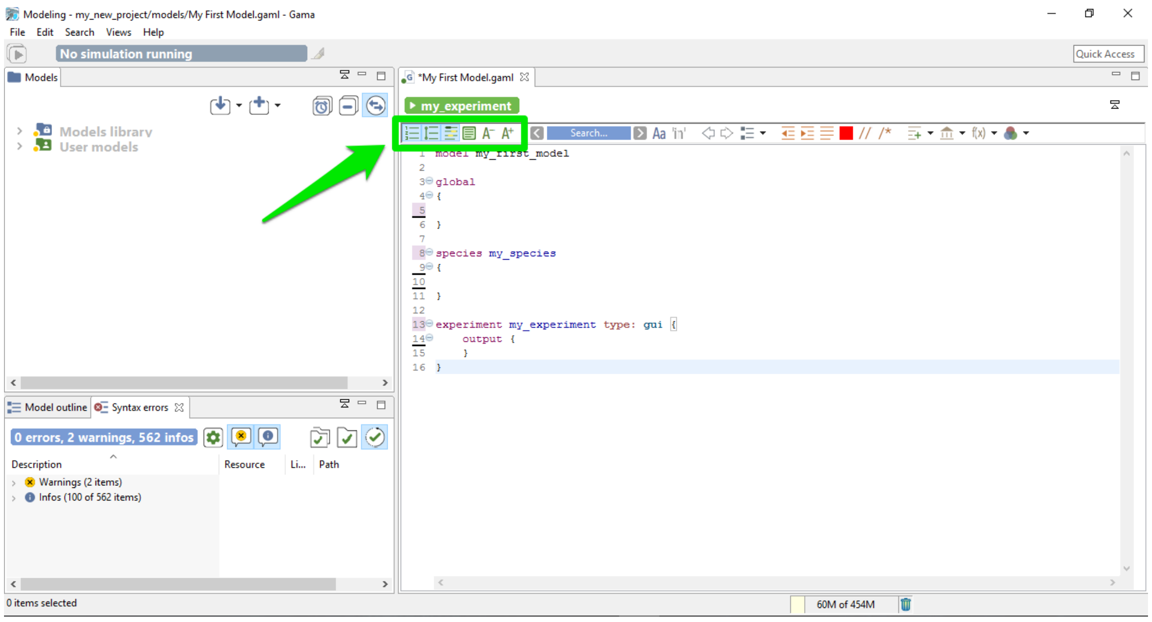
Appearance color options:

- Line number foreground
- Current line highlight
- Print margin
- Find scope
- Selection foreground color
- Selection background color
- Background color
- Foreground color
- Hyperlink

Color: ■

More colors can be configured on the 'Colors and Fonts' preference page.

Figure 10.11: images/11.editor_preferences.png



No simulation running

```
model my_first_model

global
{

}

species my_species
{

}

experiment my_experiment type: gui {
    output {
    }
}
```

0 errors, 2 warnings, 562 infos

Warnings (2 items)

Infos (100 of 562 items)

0 items selected

Figure 10.12: images/additional_informations_in_editor.png

Additional informations in the Editor

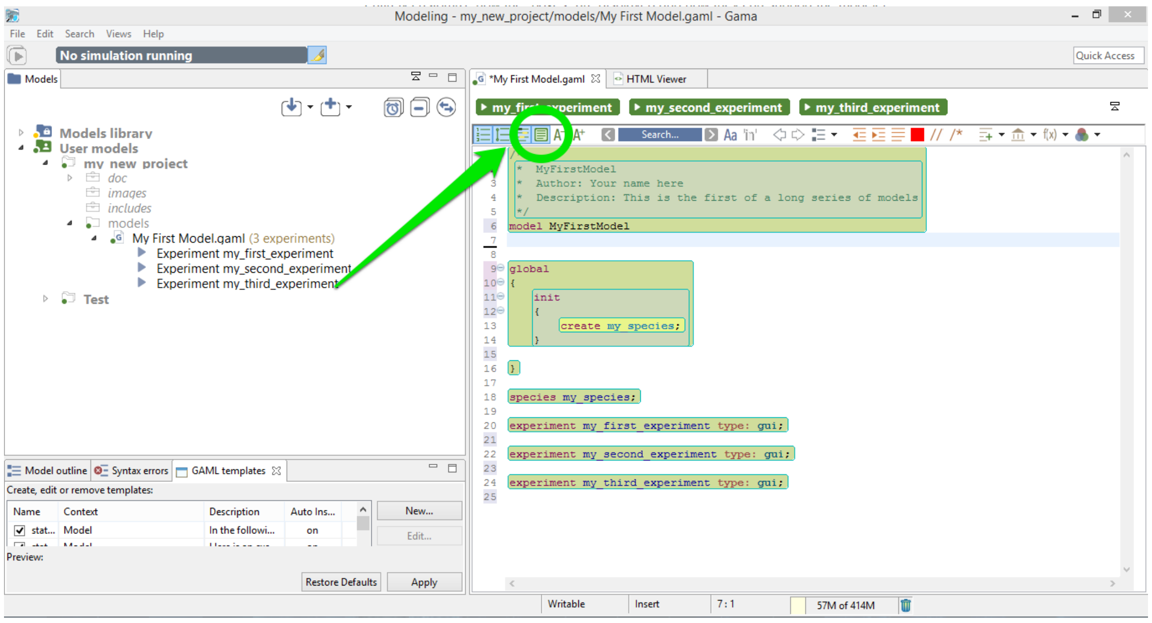
You can choose to display or not some informations in your Editor

One particular option, shipped by default with GAMA, is the possibility to not only highlight the code of your model, but also its structure (complementing, in that sense, the *Outline* view). It is a slightly modified version of a plugin called EditBox, which can be activated by clicking on the “green square” icon in the toolbar.

The Default theme of EditBox might not suit everyone’s tastes, so the preferences allow to entirely customize how the “boxes” are displayed and how they can support the modeler in better understanding “where” it is in the code. The “themes” defined in this way are stored in the workspace, but can also be exported for reuse in other workspaces, or sharing them with other modelers.

Multiple editors

GAMA inherits from Eclipse the possibility to entirely configure the placement of the views, editors, etc. This can be done by rearranging their position using the mouse (click and hold



```
*  MyFirstModel
*  Author: Your name here
*  Description: This is the first of a long series of models
*/
model MyFirstModel

global
{
	init
	{
		create my_species;
	}
}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 10.13: images/12.view_model_with_editbox_default.png

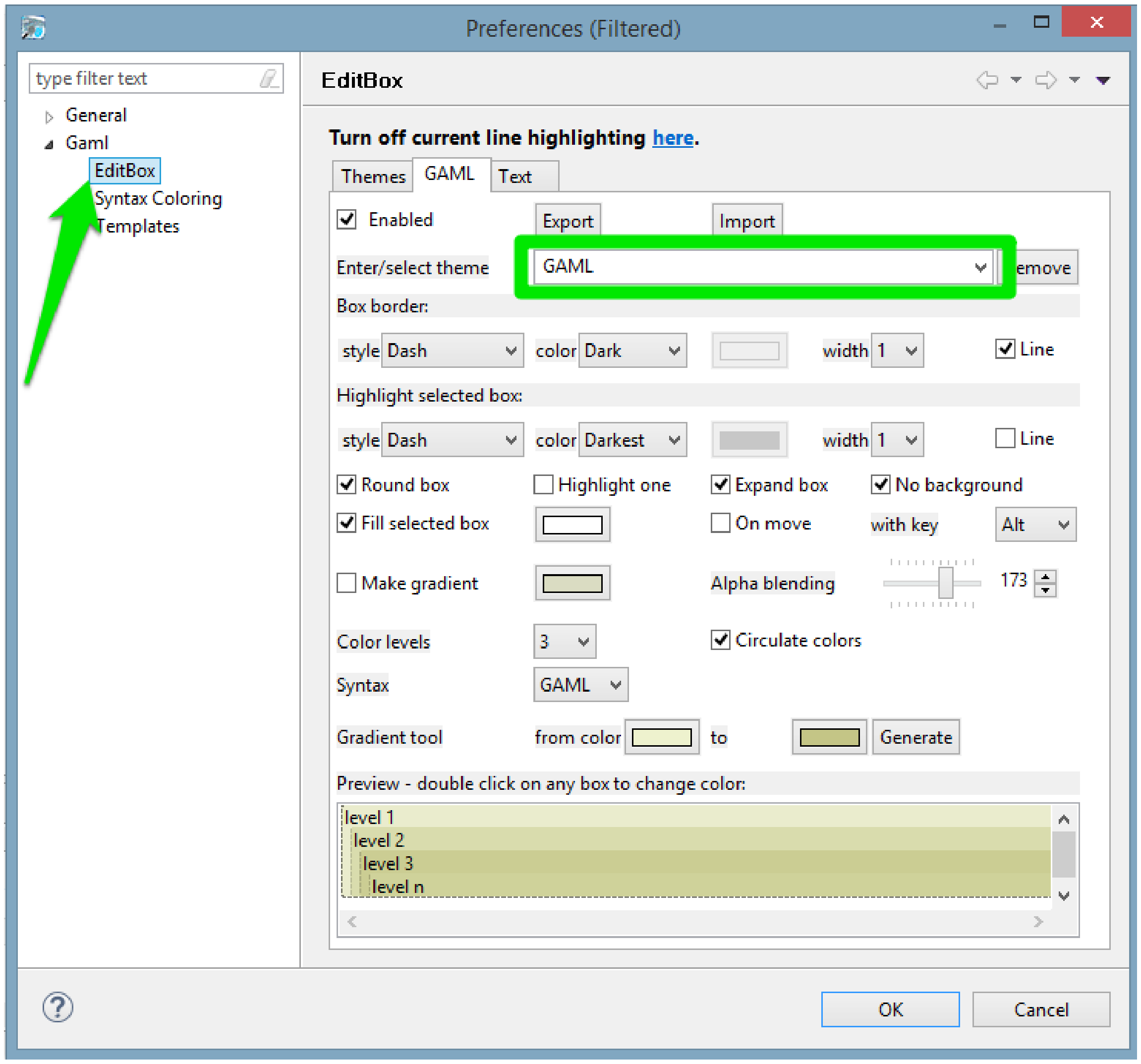
on an editor's title and move it around). In particular, you can have several editors side by side, which can be useful for viewing the documentation while coding a model.

Local history

Among the various options present to work with models, which you are invited to try out and test at will, one, called *Local history* is particularly interesting and worth a small explanation. When you edit models, GAMA keeps in the background all the successive versions you save (the history duration is configurable in the preferences), whether or not you are using a versioning system like SVN or Git. This local history is accessible from different places in GAMA (the *Navigator*, the *Views* menu, etc.), including the contextual menu of the editor.

This command invokes the opening of a new view, which you can see on the figure below, and which lists the different versions of your file so far. You can then choose one and, right-clicking on it, either open it in a new editor, or compare it to your current version.

This allows you to precisely pinpoint the modifications brought to the file and, in case of problems, to revert them easily, or even revert the entire file to a previous version. Never lose your work again !



Turn off current line highlighting here.

[✓] Enabled

Enter/select theme: GAML

Box border:

style Dash | color Dark | width 1 | [✓] Line

Highlight selected box:

style Dash | color Darkest | width 1 | [ ] Line

[✓] Round box | [ ] Highlight one | [✓] Expand box | [✓] No background

[✓] Fill selected box | [ ] On move | with key Alt

[ ] Make gradient | Alpha blending 173

Color levels 3 | [✓] Circulate colors

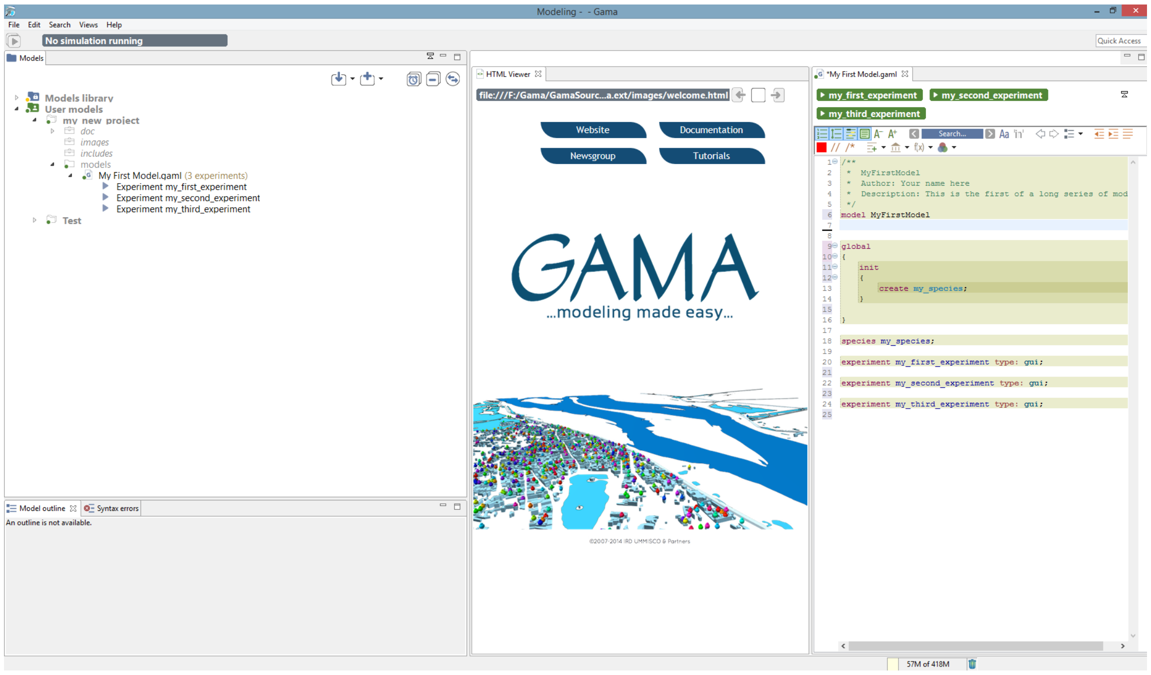
Syntax GAML

Gradient tool from color to

Preview - double click on any box to change color:

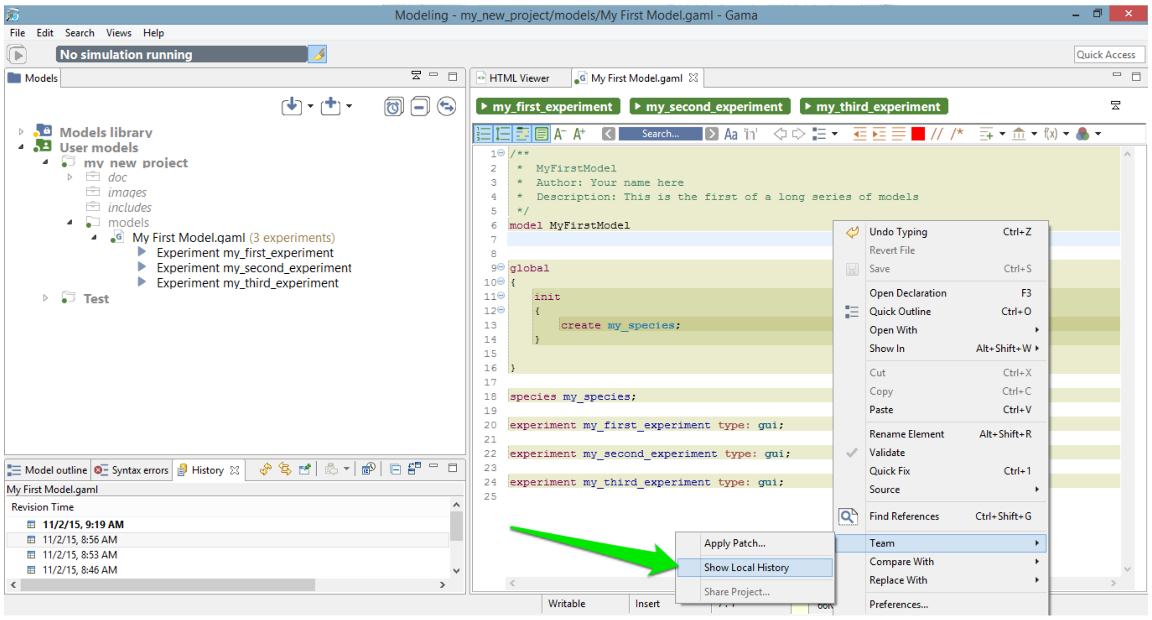
level 1
level 2
level 3
level n

Figure 10.14: images/13.editbox_preferences.png



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of mod
 */
model MyFirstModel

global
{
	init
	{
		create my_species;
	}
}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 10.15: images/14.view_model_side_by_side.png

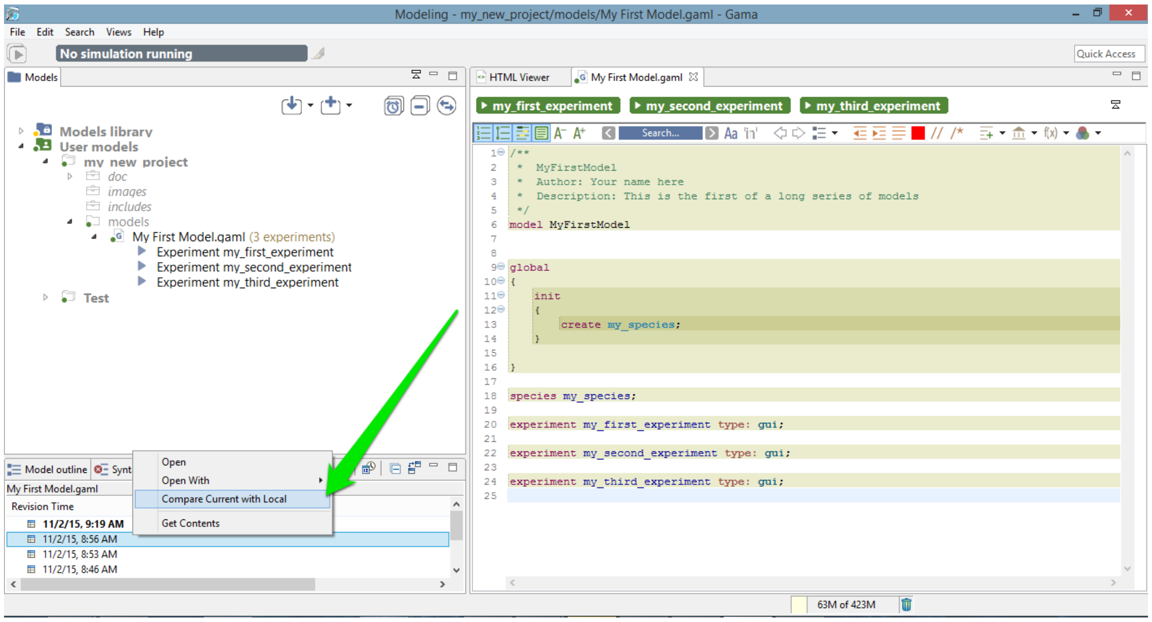


My First Model.gaml

Revision Time
11/2/15, 9:19 AM
11/2/15, 8:56 AM
11/2/15, 8:53 AM
11/2/15, 8:46 AM

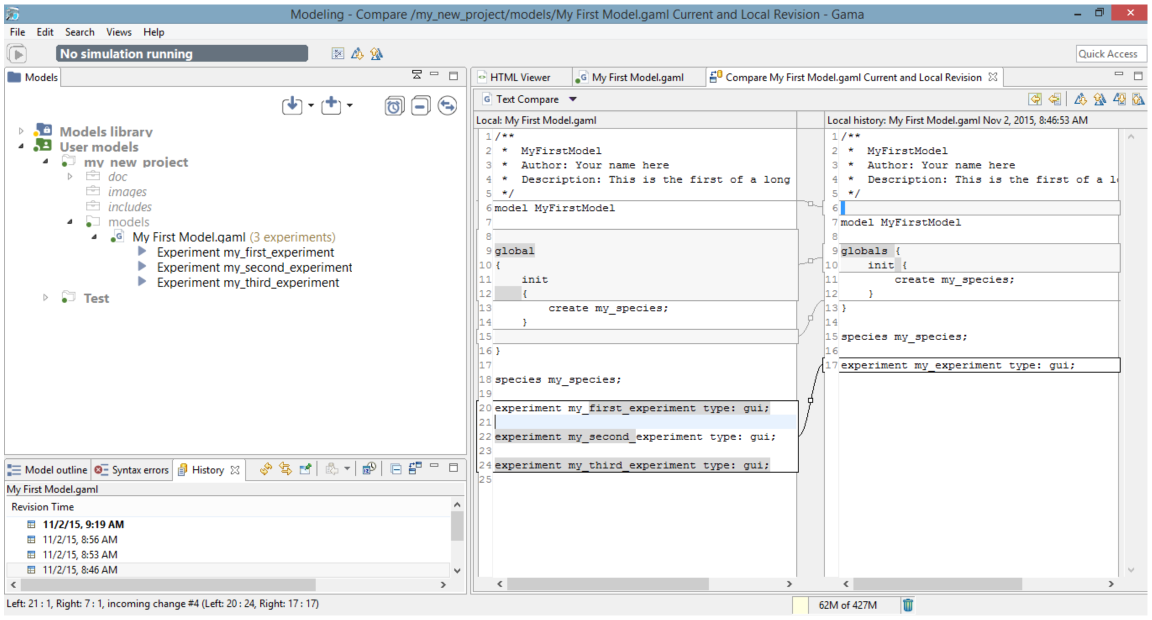
```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel

global
{
	init
	{
		create my_species;
	}
}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 10.16: images/15.view_model_with_local_history_menu.png



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel

global
{
	init
	{
		create my_species;
	}

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 10.17: images/16.view_model_with_local_history_compare_menu.png


Modeling - Compare /my_new_project/models/My First Model.gaml Current and Local Revision - Gama
File Edit Search Views Help
No simulation running
Quick Access
Models
Models library
User models
my new project
doc
images
includes
models
My First Model.gaml (3 experiments)
Experiment my_first_experiment
Experiment my_second_experiment
Experiment my_third_experiment
Test
HTML Viewer
My First Model.gaml
Compare My First Model.gaml Current and Local Revision
Text Compare
Model outline
Syntax errors
History
My First Model.gaml
Revision Time
11/2/15, 9:19 AM
11/2/15, 8:56 AM
11/2/15, 8:53 AM
11/2/15, 8:46 AM
62M of 427M

Local: My First Model.gaml

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long
 */
model MyFirstModel

global
{
	init
	{
		create my_species;
	}

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Local history: My First Model.gaml Nov 2, 2015, 8:46:53 AM

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a l
 */

model MyFirstModel

globals {
	init {
		create my_species;
	}
}

species my_species;

experiment my_experiment type: gui;
```

Left: 21 : 1, Right: 7 : 1, incoming change #4 (Left: 20 : 24, Right: 17 : 17)

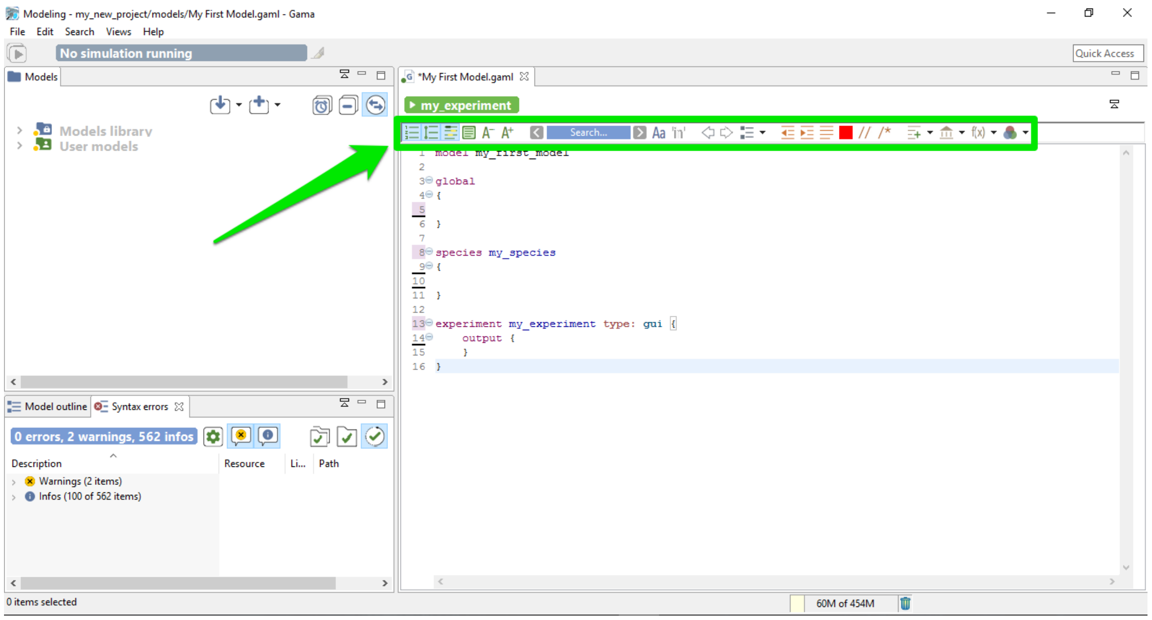
Figure 10.18: images/17.view_model_with_local_history_side_by_side.png

This short introduction to GAML editors is now over. You might want to take a look, now, at how the models you edit are parsed, validated and compiled, and how this information is accessible to the modeler.

Chapter 11

The GAML Editor Toolbar

The GAML Editor provide some tools to make the editing easier, covering a lot of functionalities, such as tools for changes of visualization, tools for navigation through your model, tools to format your code, or also tools to help you finding the correct keywords to use in a given context.

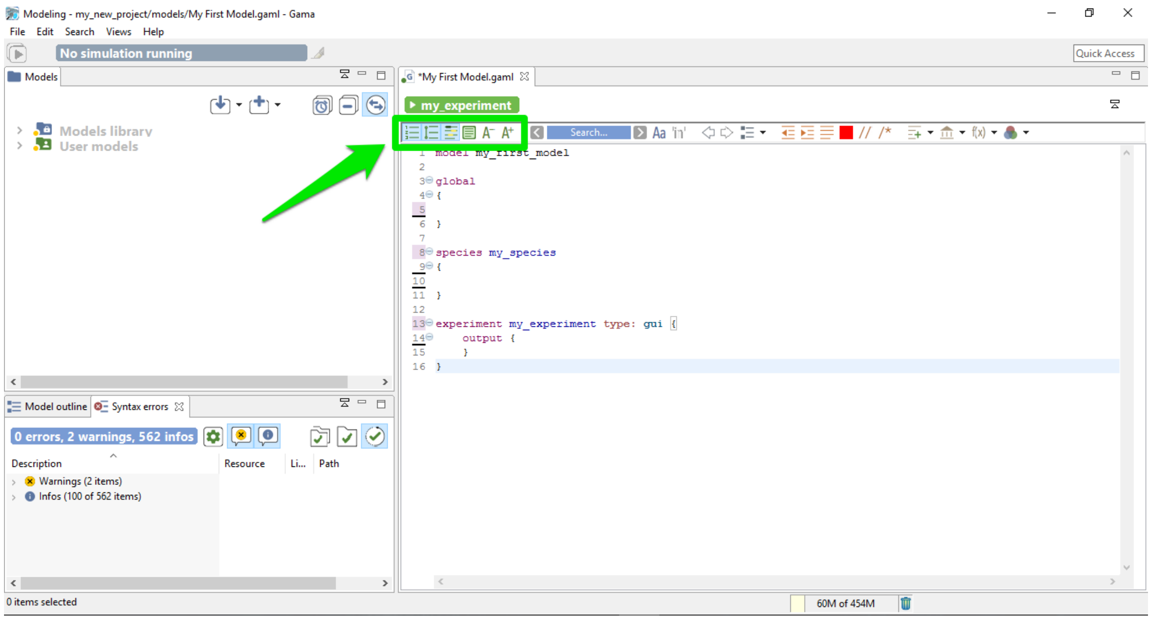


```
model my_first_model

global
{

}

species my_species
{

}

experiment my_experiment type: gui {
    output {
    }
}
```

Figure 11.1: images/graphical_editor_toolbar.png

Table of contents

Visualization tools in the editor



```
model my_first_model

global
{

}

species my_species
{

}

experiment my_experiment type: gui {
	output {
	}
}
```

Figure 11.2: images/additional_informations_in_editor.png

You can choose to display or not some informations in your Editor. Here are the different features for this part:

Display the number of lines

The first toggle is used to show / hide the number of lines.

Expand / Collapse lines

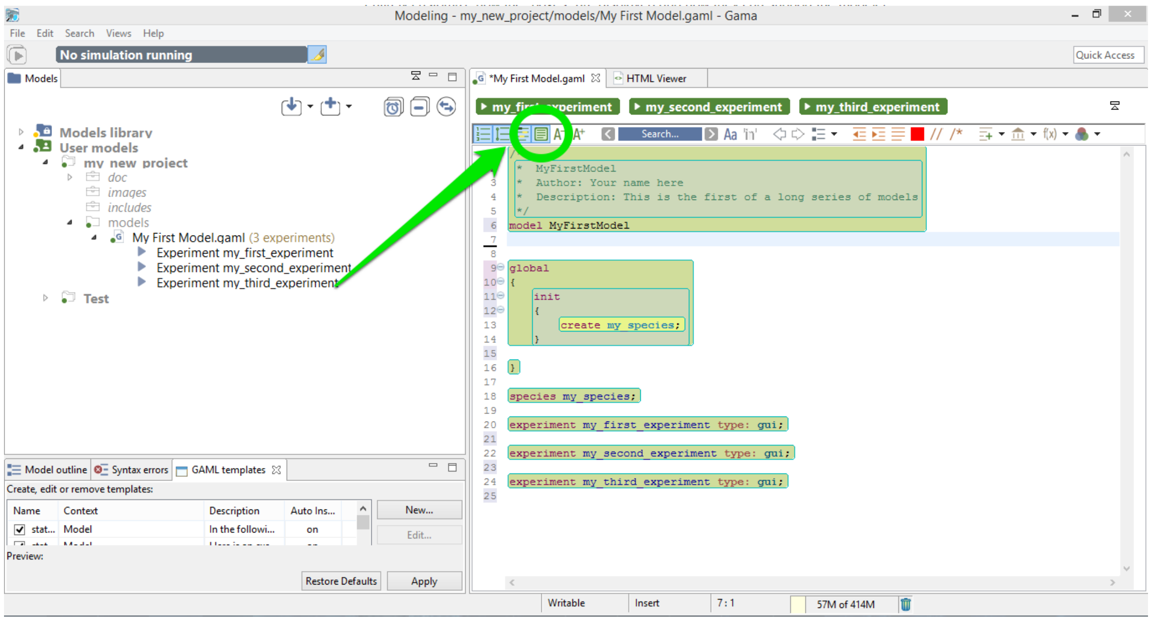
The second toggle provides you the possibility to expand or collapse lines in your model depending on the indentation. This feature can be very useful for big models, to collapse the part you have already finished.

Mark the occurrences

This third toggle is used to show occurrences when your cursor is pointing on one word.

Display colorization of code section

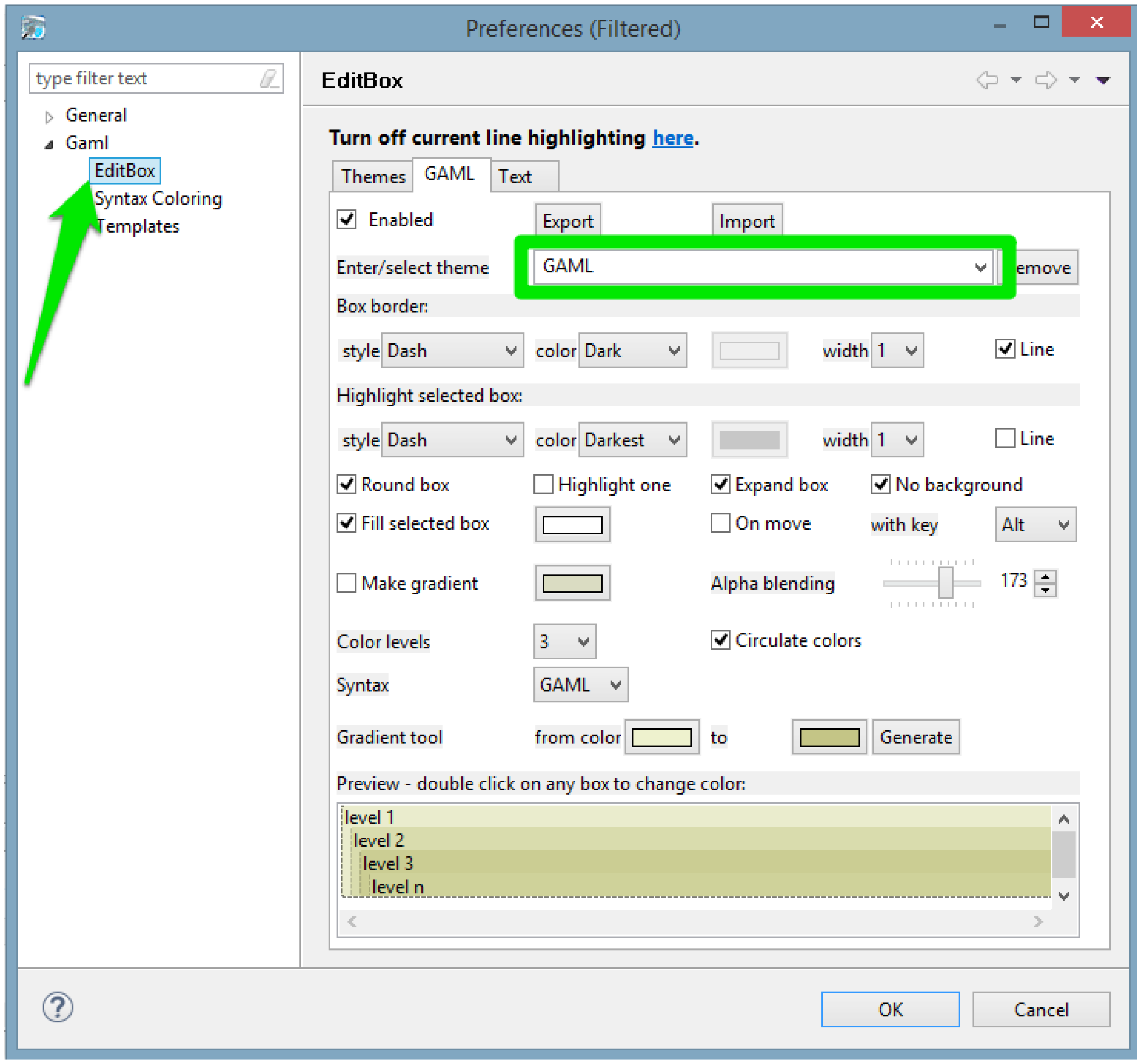
One particular option, shipped by default with GAMA, is the possibility to not only highlight the code of your model, but also its structure (complementing, in that sense, the *Outline* view). It is a slightly modified version of a plugin called EditBox, which can be activated by clicking on the "green square" icon in the toolbar.



```
/
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        create my_species;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 11.3: images/12.view_model_with_editbox_default.png

The Default theme of EditBox might not suit everyone's tastes, so the preferences allow to entirely customize how the "boxes" are displayed and how they can support the modeler in better understanding "where" it is in the code. The "themes" defined in this way are stored in the workspace, but can also be exported for reuse in other workspaces, or sharing them with other modelers.


Preferences (Filtered)

type filter text

- General
- Gaml
  - EditBox
  - Syntax Coloring
  - Templates

EditBox

Turn off current line highlighting here.

Themes | GAML | Text

☑ Enabled Export Import

Enter/select theme: GAML emove

Box border:
style Dash, color Dark, width 1, ☑ Line

Highlight selected box:
style Dash, color Darkest, width 1, ☐ Line

☑ Round box ☐ Highlight one ☑ Expand box ☑ No background

☑ Fill selected box ☐ On move with key Alt

☐ Make gradient Alpha blending 173

Color levels 3 ☑ Circulate colors

Syntax GAML

Gradient tool from color to Generate

Preview - double click on any box to change color:

level 1
level 2
level 3
level n

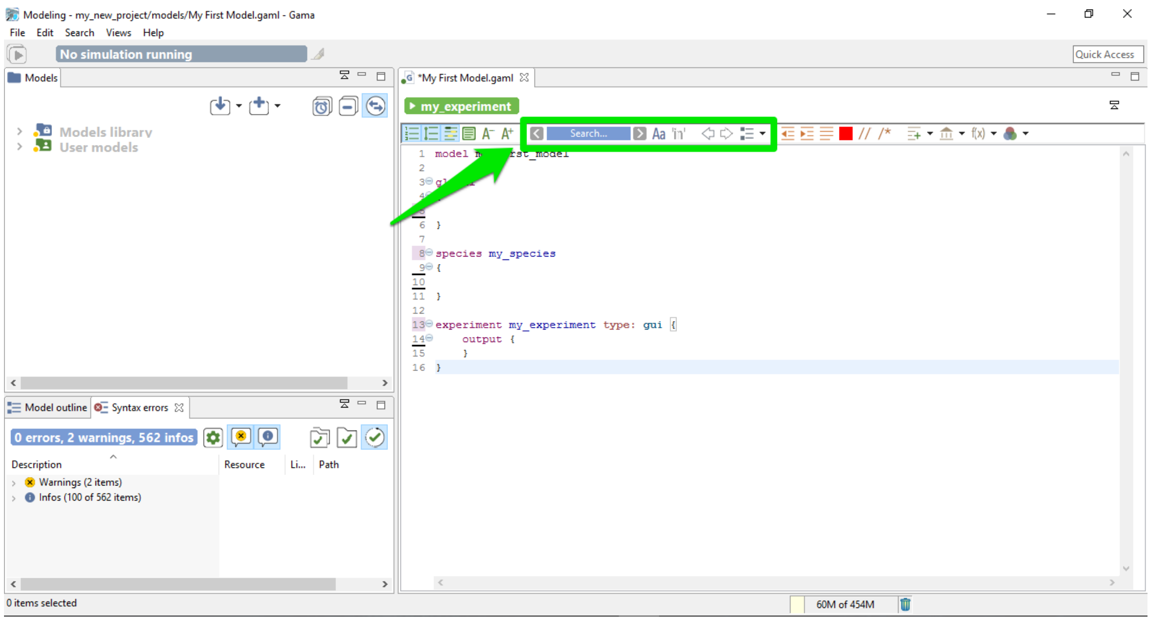
OK Cancel

Figure 11.4: images/13.editbox_preferences.png

Change the font size

The two last tools of this section are used to increase / decrease the size of the displayed text.

Navigation tools in the editor



```
model m[illegible]rst_model

gl[illegible]
[illegible]

}

species my_species
{

}

experiment my_experiment type: gui {
    output {
    }
}
```

0 errors, 2 warnings, 562 infos

Warnings (2 items)
Infos (100 of 562 items)

Figure 11.5: images/navigation_in_editor.png

In the Editor toolbar, you have some tools for search and navigation through the code. Here are the explanation for each functionalities:

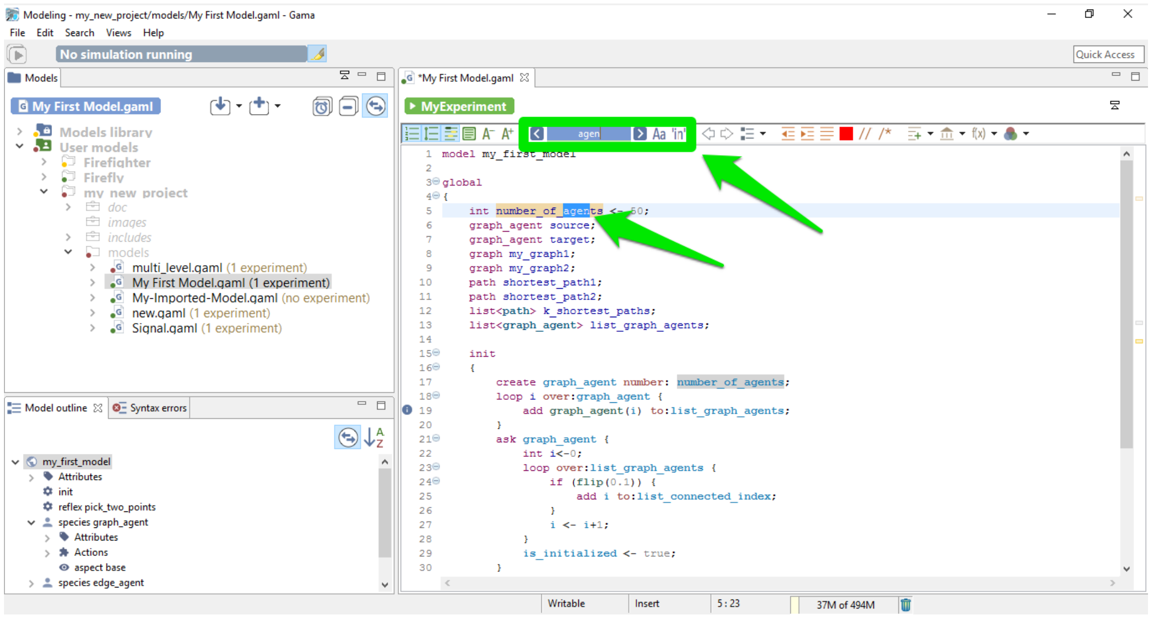
The search engine

In order to search an occurrence of a word (or the part of a word), you can type your search in the field, and the result will be highlighted automatically in the text editor.

With the left / right arrows, you can highlight the previous / next occurrence of the word. The two toggles just in the right side of the search field are used to constraint the results as “case sensitive” or “whole word”. If you prefer the eclipse interface for the search engine, you can also access to the tool by taping Ctrl+F.

Previous / Next location

The two arrow shape buttons that are coming after are used to jump from the current location of your cursor to the last position, even if the last position was in an other file (and even if



```
model my_first_model

global
{
    int number_of_agents <- 50;
    graph_agent source;
    graph_agent target;
    graph my_graph1;
    graph my_graph2;
    path shortest_path1;
    path shortest_path2;
    list<path> k_shortest_paths;
    list<graph_agent> list_graph_agents;

    init
    {
        create graph_agent number: number_of_agents;
        loop i over:graph_agent {
            add graph_agent(i) to:list_graph_agents;
        }
        ask graph_agent {
            int i<-0;
            loop over:list_graph_agents {
                if (flip(0.1)) {
                    add i to:list_connected_index;
                }
                i <- i+1;
            }
            is_initialized <- true;
        }
```

Figure 11.6: images/search_engine.png

this file has been closed !).

Show outline

This last tool of this section is used to show the global architecture of your model, with explicit icons for each section. A search field is also available, if you want to search a specific section. By double clicking one line of the outline, you can jump directly to the chosen section. This feature can be useful if you have big model to manipulate.

Format tools in the editor

Some other tools are available in the toolbar to help for the indentation of the model:

Shift left / shift right

Those two first buttons are used to shift a line (or a group of lines) on the left or the right.

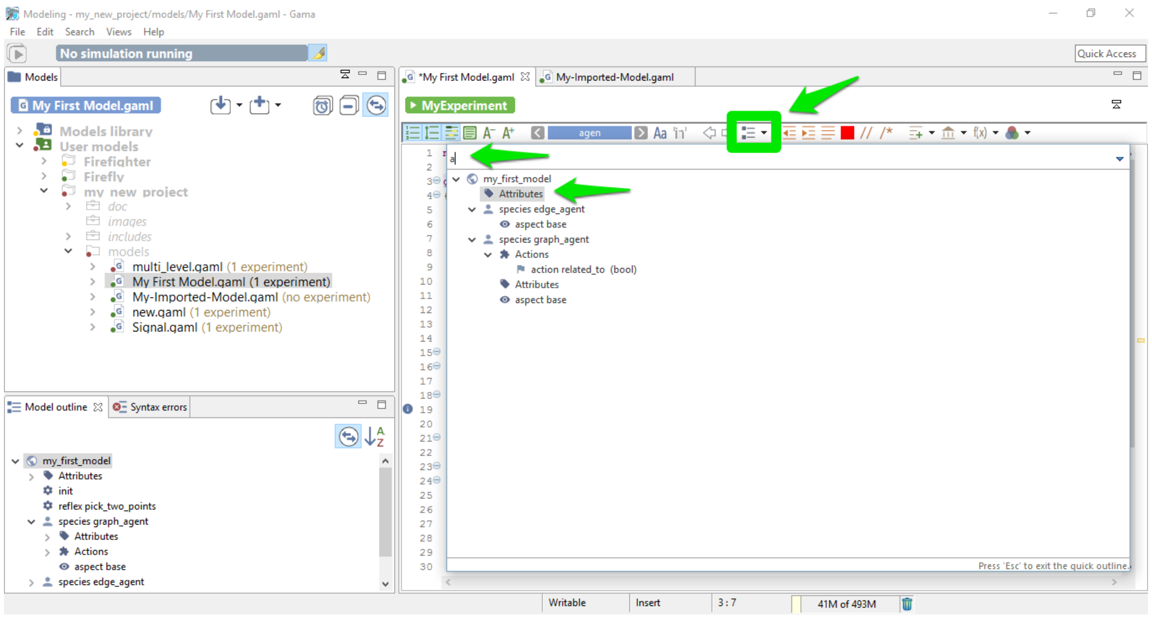
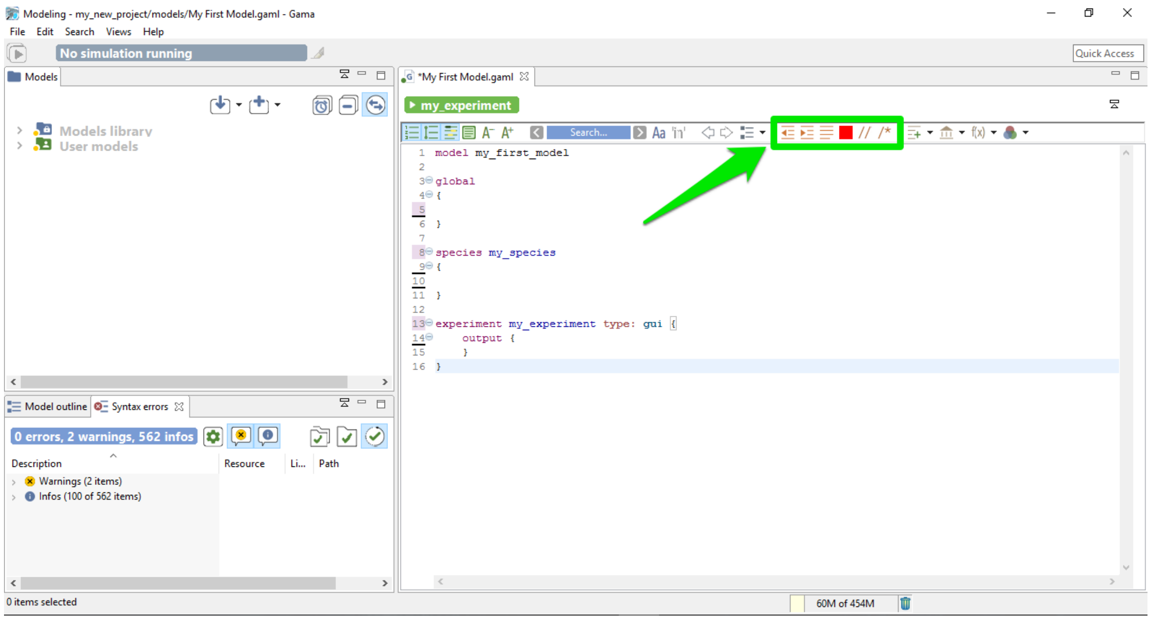


Figure 11.7: images/show_outline.png



```
model my_first_model

global
{

}

species my_species
{

}

experiment my_experiment type: gui {
    output {
    }
}
```

Figure 11.8: images/format_the_text_in_editor.png

Format

This useful feature re-indent automatically all your model.

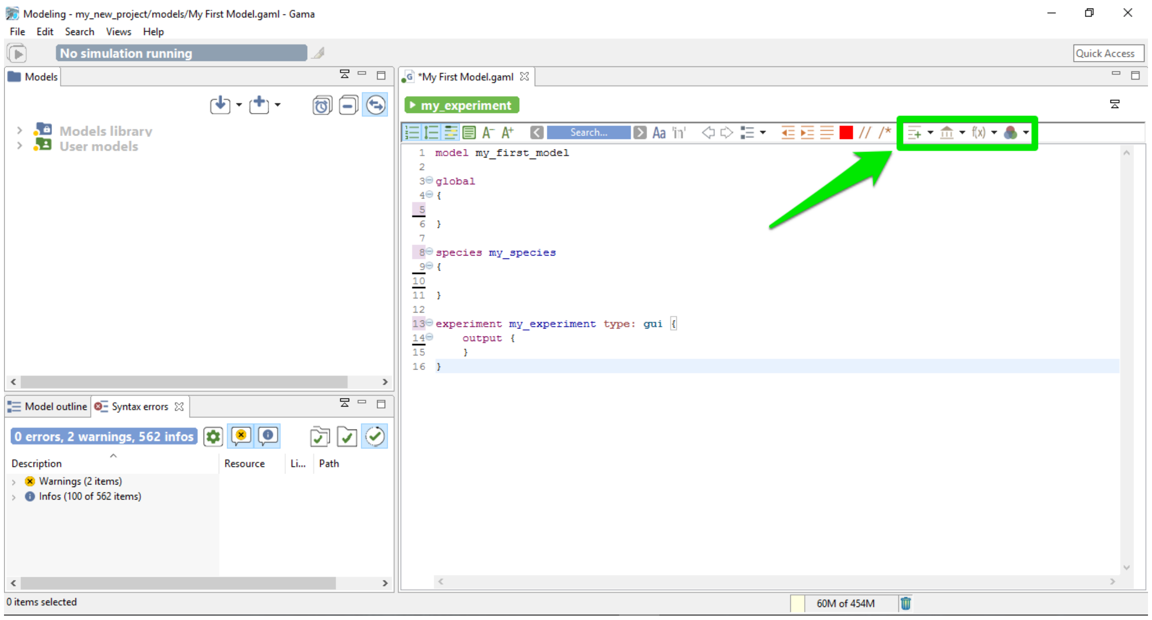
Re-serialize

Re-serialize your model.

Comment

The two last buttons of this section are useful to comment a line (or a group of lines).

Vocabulary tools in the editor



```
model my_first_model

global
{

}

species my_species
{

}

experiment my_experiment type: gui {
    output {
    }
}
```

Figure 11.9: images/vocabulary_help_in_editor.png

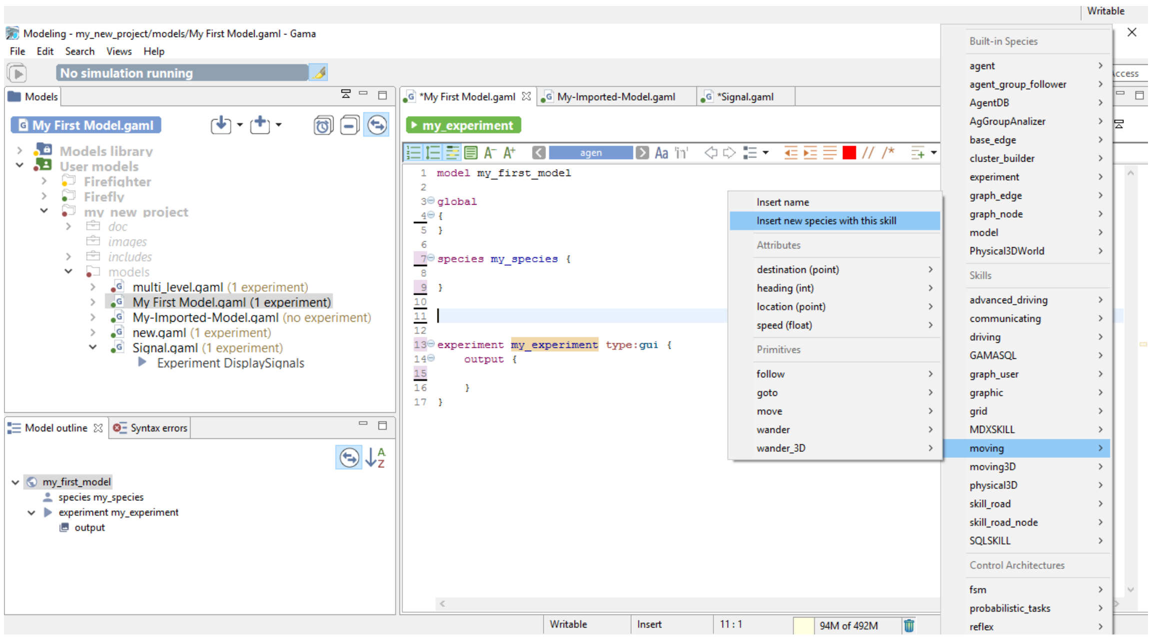
The last group of buttons are used to search the correct way to write a certain keyword.

Templates

The templates button is used to insert directly a code snippet in the current position of the cursor. Some snippets are already available, ordered by scope. You can custom the list of template as much as you want, it is very easy to add a new template.

Built-in attributes, built-in actions

With this feature, you can easily know the list of built-in attributes and built-in actions you can use in such or such context. With this feature, you can also insert some templates to help you, for example to insert a pre-made species using a particular skill, as it is shown it the following screenshot:

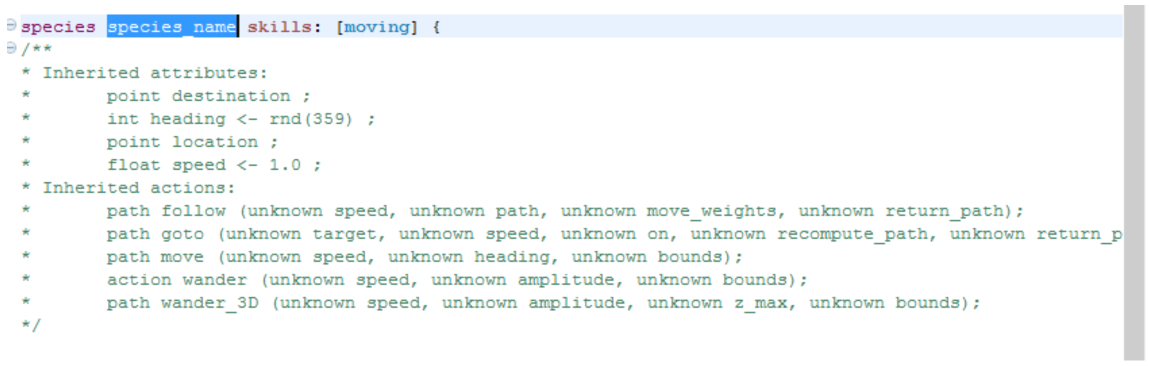


```
model my_first_model

global
{
}

species my_species {

}

experiment my_experiment type:gui {
    output {

    }
}
```

Figure 11.10: images/insert_species_with_moving_skill1.png

... will generate the following code:

All the comments are generated automatically from the current documentation.



```
species species_name skills: [moving] {
/**
 * Inherited attributes:
 *       point destination ;
 *       int heading <- rnd(359) ;
 *       point location ;
 *       float speed <- 1.0 ;
 * Inherited actions:
 *       path follow (unknown speed, unknown path, unknown move_weights, unknown return_path);
 *       path goto (unknown target, unknown speed, unknown on, unknown recompute_path, unknown return_p
 *       path move (unknown speed, unknown heading, unknown bounds);
 *       action wander (unknown speed, unknown amplitude, unknown bounds);
 *       path wander_3D (unknown speed, unknown amplitude, unknown z_max, unknown bounds);
 */
```

Figure 11.11: images/insert_species_with_moving_skill2.png

Operators

Once again, this powerful feature is used to generate example of structures for all the operators, ordered by categories.

Colors

Here is the list of the name for the different pre-made colors you can use. You can also add some custom colors.

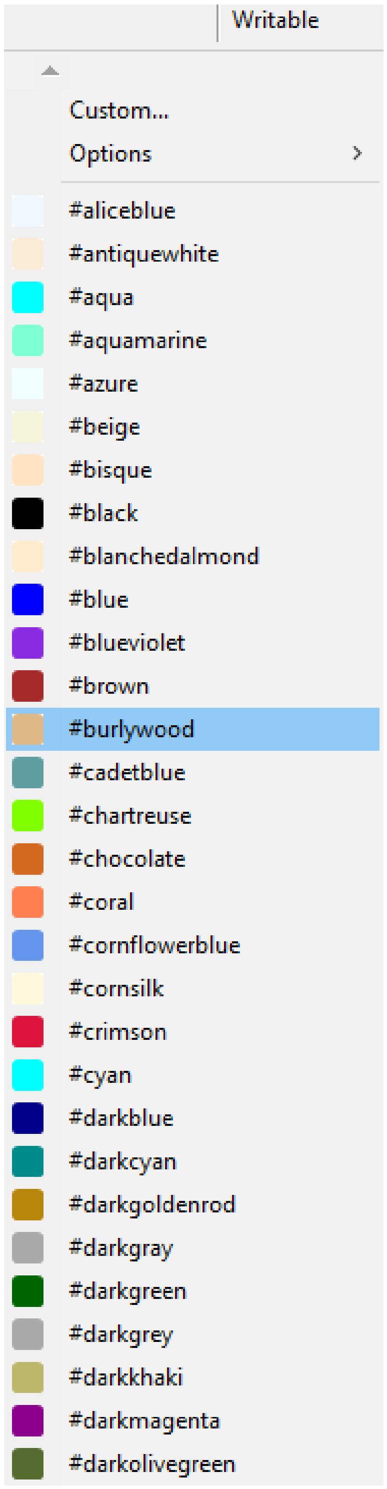


Figure 11.12: images/color.png

Chapter 12

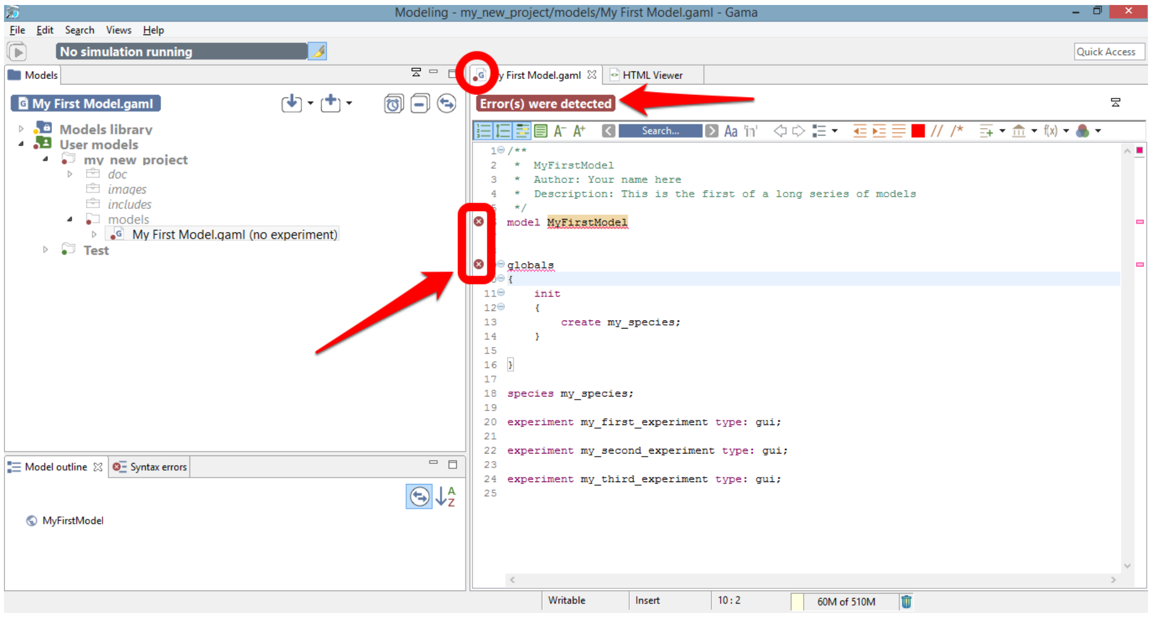
Validation of Models

When editing a model, GAMA will continuously validate (i.e. *compile*) what the modeler is entering and indicate, with specific visual affordances, various information on the state of the model. This information ranges from documentation items to errors indications. We will review some of them in this section.

Table of contents

Syntactic errors

These errors are produced when the modeler enters a sentence that has no meaning in the grammar of GAML (see the documentation of the language). It can either be a non-existing symbol (like "globals" (instead of *global*) in the example below), a wrong punctuation scheme, or any other construct that puts the parser in the incapacity of producing a correct syntax tree. These errors are extremely common when editing models (since incomplete keywords or sentences are continuously validated). GAMA will report them using several indicators: the icon of the file in the title of the editor will sport an error icon and the gutter of the editor (i.e. the vertical space beside the line numbers) will use error **markers** to report two or more errors: one on the statement defining the model, and one (or more) in the various places where the parser has failed to produce the syntax tree. In addition, the toolbar over the editor will turn red and indicate that errors have been detected.



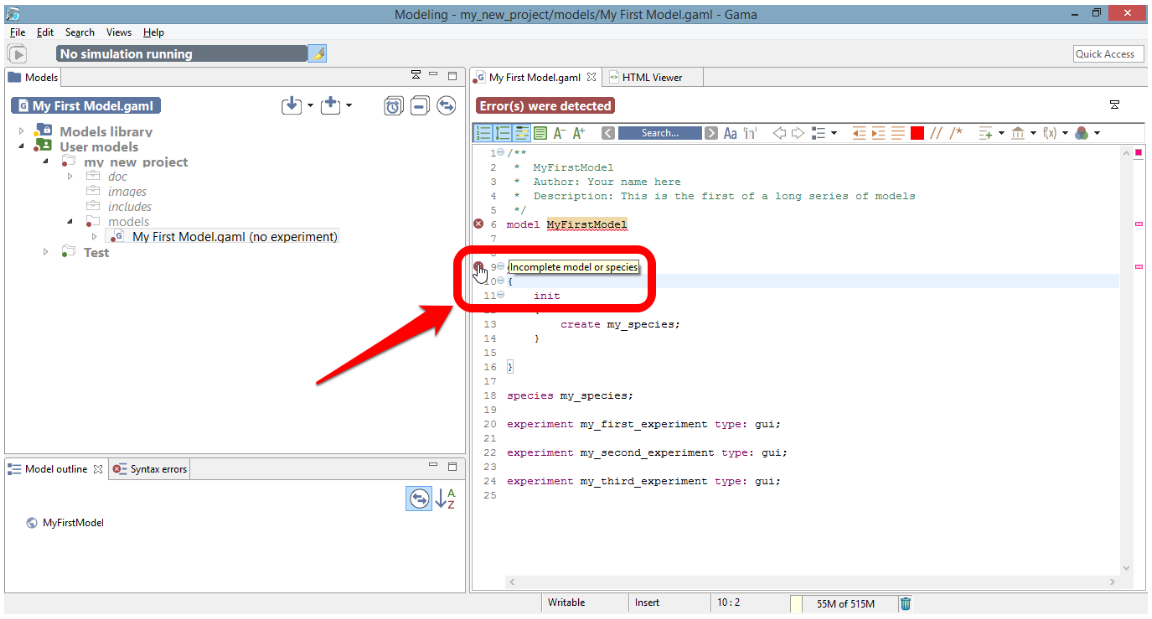
```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


globals
{
    init
    {
        create my_species;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 12.1: images/model_with_syntactic_errors.png

Hovering over one of these **markers** indicates what went wrong during the syntactic validation. Note that these errors are sometimes difficult to interpret, since the parser might fail in places that are not precisely those where a wrong syntax is being used (it will usually fail **after**).

Semantic errors

When syntactic errors are eliminated, the validation enters a so-called semantic phase, during which it ensures that what the modeler has written makes sense with respect to the vari-



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel

{
    init

        create my_species;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

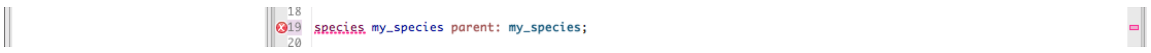
Figure 12.2: images/model_with_syntactic_errors_and_hover.png

ous rules of the language. To understand the difference between the two phases, take a look at the following example.

This sentence below is **syntactically** correct:

```
species my_species parent: my_species;
```

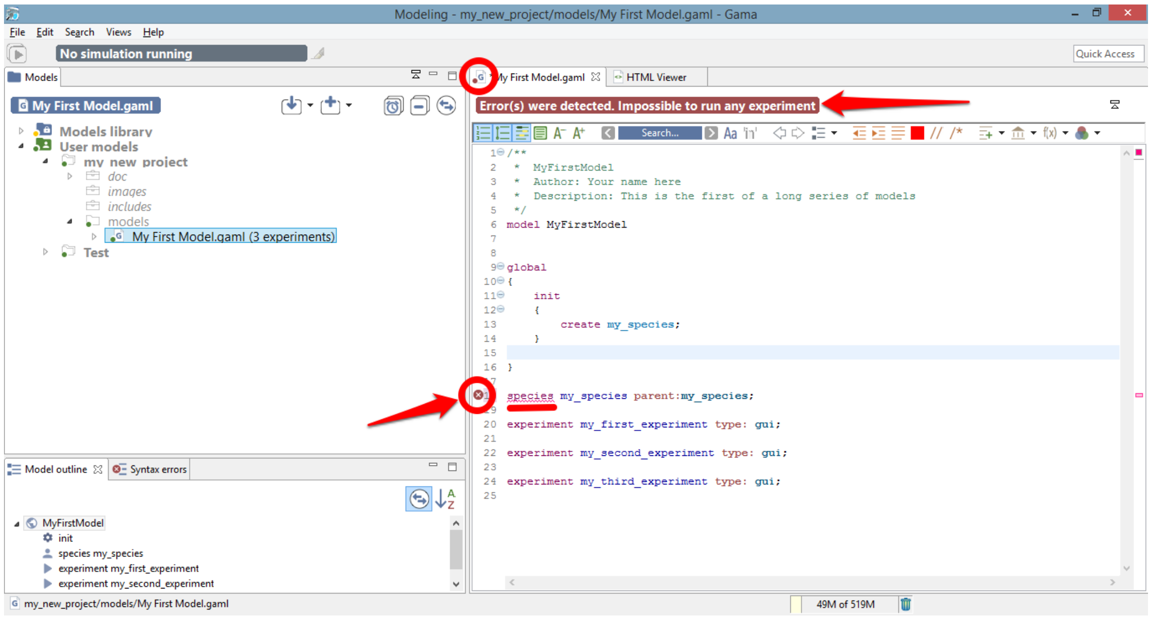
But it is **semantically** incorrect because a species cannot be parent of itself. No syntactic errors will be reported here, but the validation will fail with a **semantic** error.



```
species my_species parent: my_species;
```

Figure 12.3: images/semantic_error_detail.png

Semantic errors are reported in a way similar to syntactic errors, except that no **marker** are displayed beside the model statement. The compiler tries to report them as precisely as possible, underlining the places where they have been found and outputting hopefully meaningful error messages. In the example below, for instance, we use a wrong number of arguments for defining a square geometry. Although the sentence is syntactically correct, GAMA will nevertheless issue an error and prevent the model from being experimentable.



Error(s) were detected. Impossible to run any experiment

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        create my_species;
    }

}

species my_species parent:my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;

```

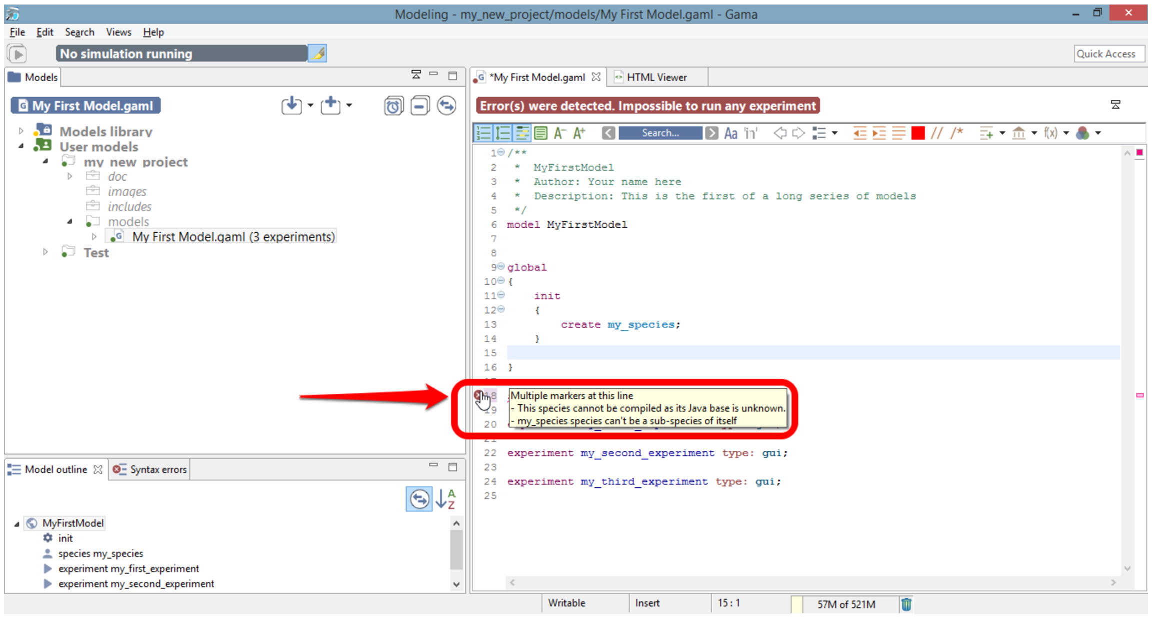
Figure 12.4: images/model_with_semantic_errors.png

The message accompanying this error can be obtained by hovering over the error **marker** found in the gutter (multiple messages can actually be produced for a same error, see below).

While the editor is in a so-called *dirty* state (i.e. the model has not been saved), errors are only reported locally (in the editor itself). However, as soon as the user saves a model containing syntactic or semantic errors, they are "promoted" to become workspace errors, and, as such, indicated in other places: the file icon in the *Navigator*, and a new line in the *Errors* view.

Semantic warnings

The semantic validation phase does not only report errors. It also outputs various indicators that can help the modeler in verifying the correctness of his/her model. Among them are **warnings**. A warning is an indication that something is not completely right in the way the model is written, although it can *probably* be worked around by GAMA when the model will be executed. For instance, in the example below, we pass a string argument to the facet "number:" of the "create" statement. GAMA will emit a warning in such a case, indicating that "number:" expects an integer, and that the string passed will be casted to int when the model will be executed. Warnings are to be considered seriously, as they usually indicate



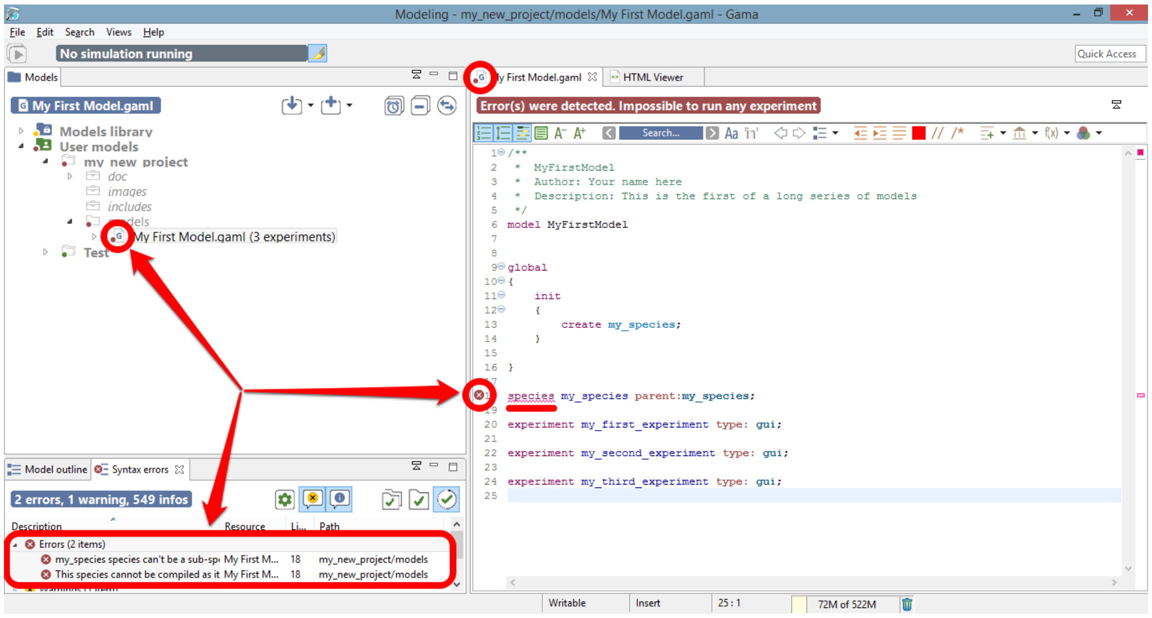
Error(s) were detected. Impossible to run any experiment

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        create my_species;
    }

}
```

Multiple markers at this line
- This species cannot be compiled as its Java base is unknown.
- my_species species can't be a sub-species of itself

```
experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 12.5: images/model_with_semantic_errors_and_hover.png



Error(s) were detected. Impossible to run any experiment

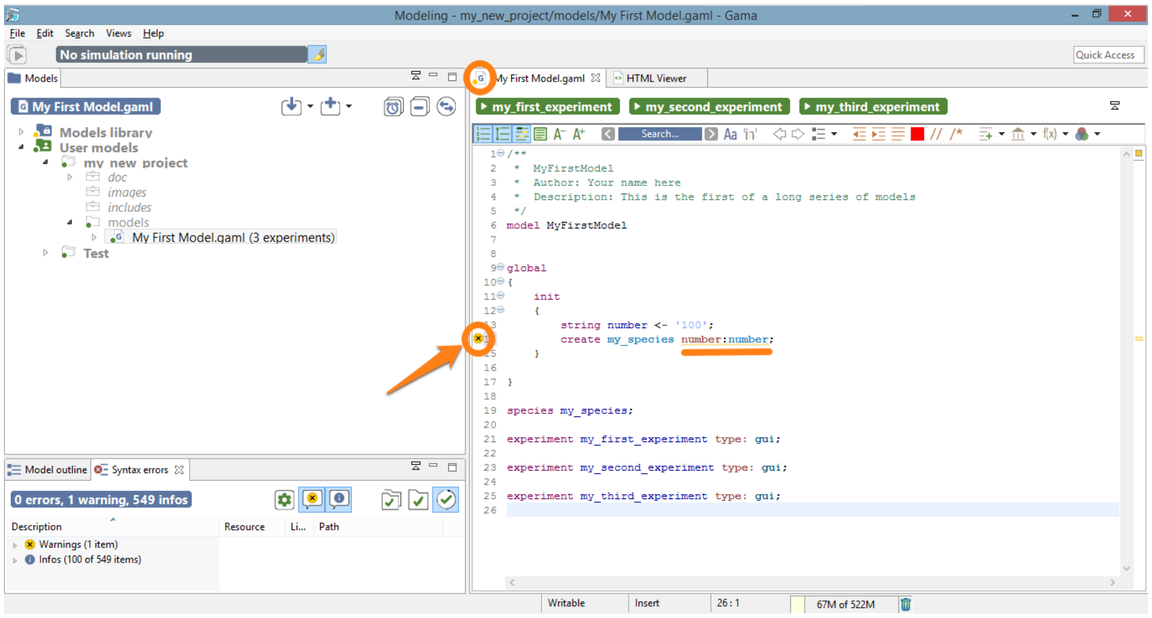
```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        create my_species;
    }

}

species my_species parent:my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

2 errors, 1 warning, 549 infos

Description	Resource	Li...	Path
Errors (2 items)			
my_species species can't be a sub-spe	My First M...	18	my_new_project/models
This species cannot be compiled as it	My First M...	18	my_new_project/models

Figure 12.6: images/model_with_semantic_errors_saved.png

some flaws in the logic of the model.



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        string number <- '100';
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

0 errors, 1 warning, 549 infos

Warnings (1 item)
Infos (100 of 549 items)

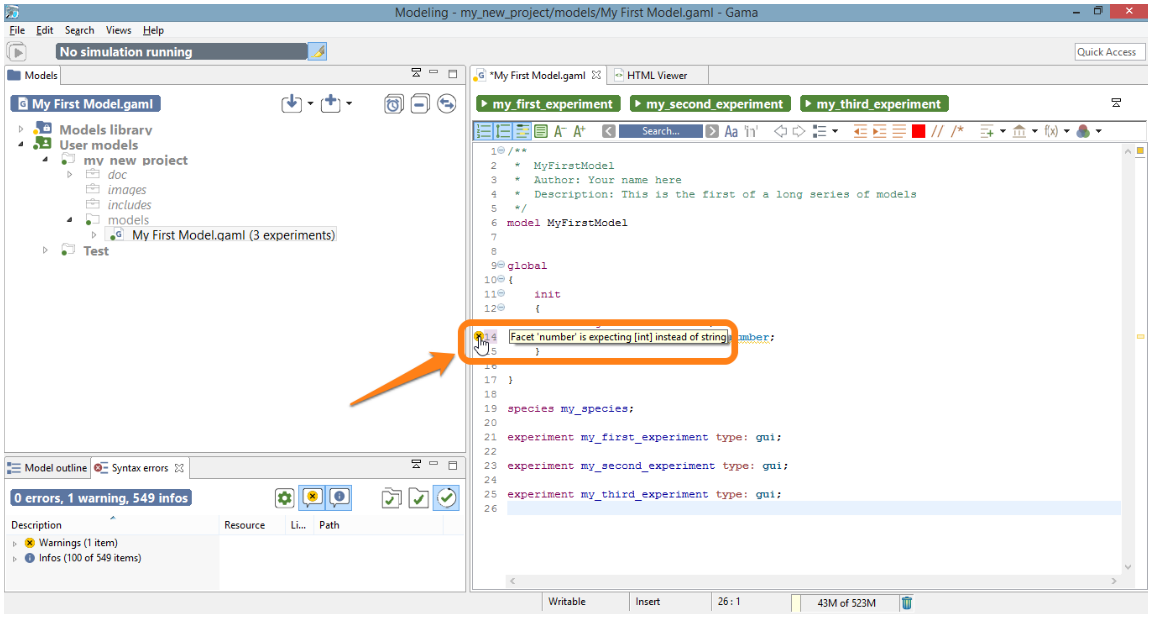
Figure 12.7: images/model_with_warnings.png

Hovering over the warning **marker** will allow the modeler to have access to the explanation and hopefully fix the cause of the warning.

Semantic information

Besides warnings, another type of harmless feedback is produce by the semantic validation phase: information **markers**. They are used to indicate useful information to the modeler, for example that an attribute has been redefined in a sub-species, or that some operation will take place when running the model (for instance, the truncation of a float to an int). The visual affordance used in this case is voluntarily discrete (a small "i" in the editor's gutter).

As with the other types of **markers**, information markers unveil their messages when being hovered.



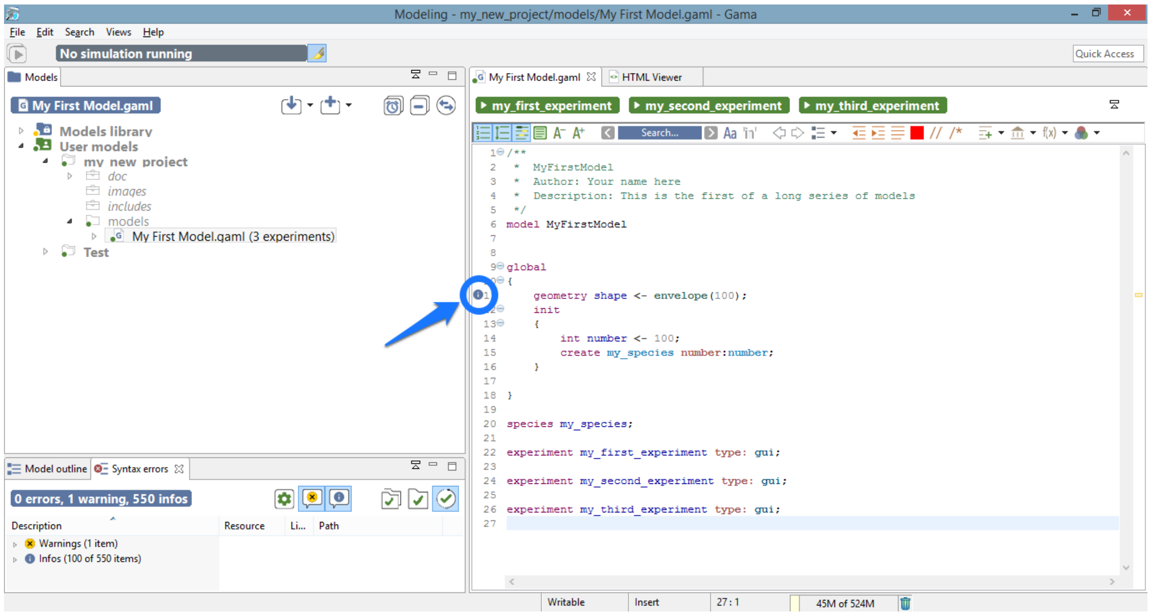
Facet 'number' is expecting [int] instead of string

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
                                                  umber;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

0 errors, 1 warning, 549 infos

Description	Resource	Li...	Path
Warnings (1 item)			
Infos (100 of 549 items)			

Figure 12.8: images/model_with_warnings_and_hover.png

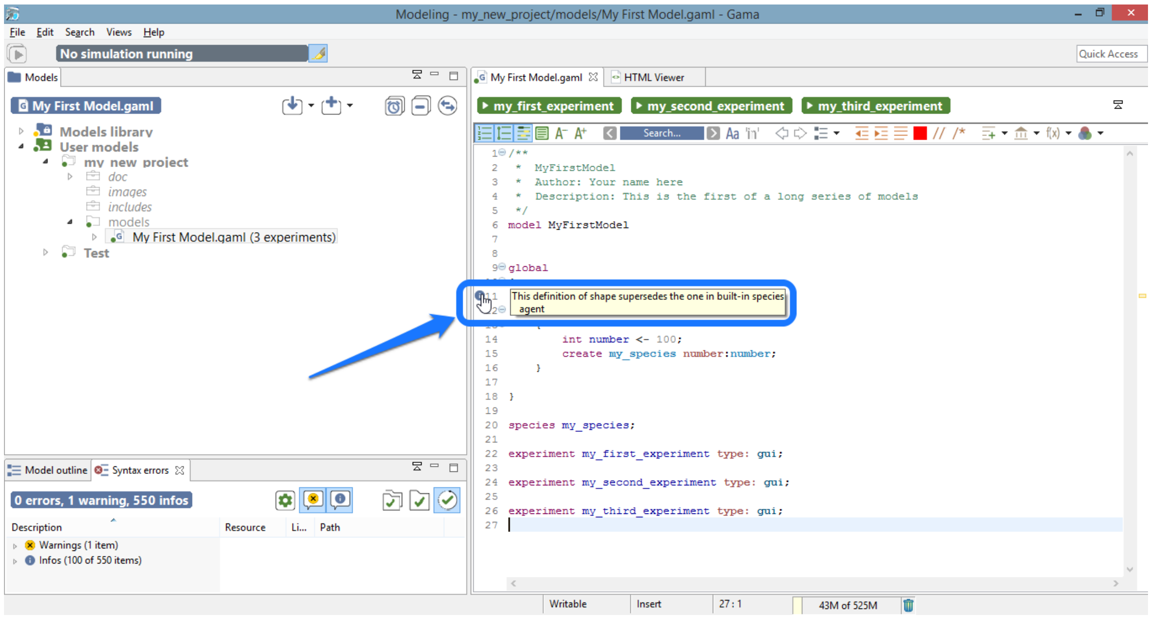


```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    geometry shape <- envelope(100);
    init
    {
        int number <- 100;
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

0 errors, 1 warning, 550 infos

Description	Resource	Li...	Path
Warnings (1 item)			
Infos (100 of 550 items)			

Figure 12.9: images/model_with_info.png



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global

        int number <- 100;
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

This definition of shape supersedes the one in built-in species agent

0 errors, 1 warning, 550 infos

Warnings (1 item)

Infos (100 of 550 items)

Figure 12.10: images/model_with_info_and_hover.png

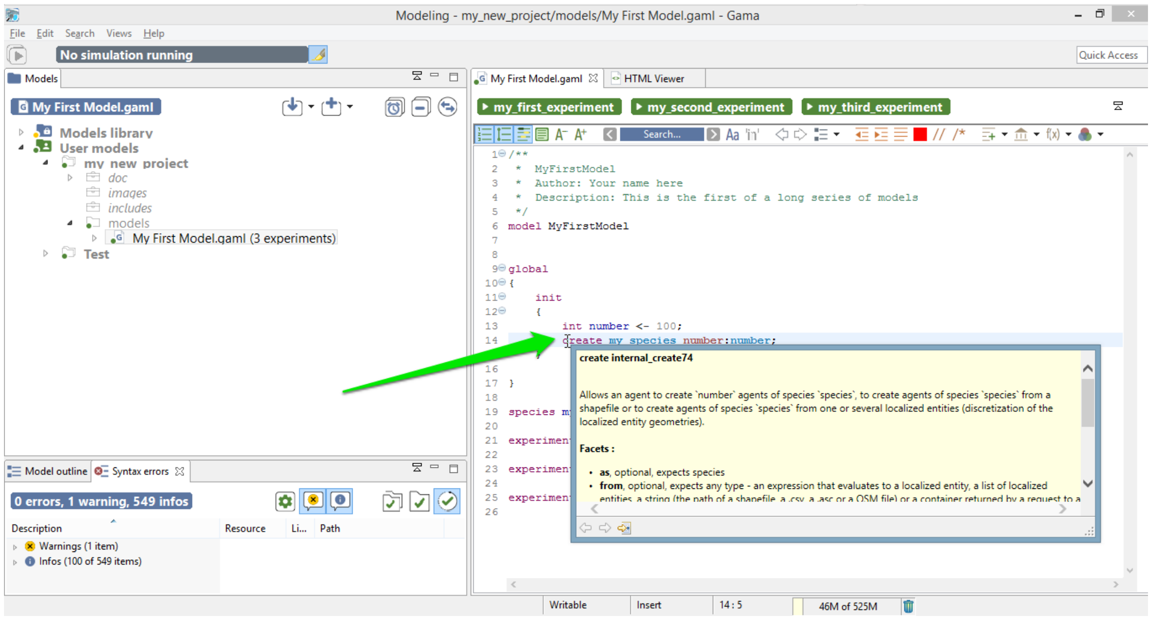
Semantic documentation

The last type of output of the semantic validation phase consists in a complete documentation of the various elements present in the model, which the user can retrieve by hovering over the different symbols. Note that although the best effort is being made in producing a complete and consistent documentation, it may happen that some symbols do not produce anything. In that case, please report a new Issue here.

Changing the visual indicators

The default visual indicators depicted in the examples above to report errors, warnings and information can be customized to be less (or more) intrusive. This can be done by choosing the "Preferences..." item of the editor contextual menu and navigating to "General > Editors > Text Editors > Annotations". There, you will find the various **markers** used, and you will be able to change how they are displayed in the editor's view. For instance, if you prefer to highlight errors in the text, you can change it here.

Which will result in the following visual feedback for errors:



```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        int number <- 100;
        create my_species number:number;


}

species m

experimen

experimen

experimen

```

create internal_create74

Allows an agent to create `number` agents of species `species`, to create agents of species `species` from a shapefile or to create agents of species `species` from one or several localized entities (discretization of the localized entity geometries).

Facets :

- **as**, optional, expects species
- **from**, optional, expects any type - an expression that evaluates to a localized entity, a list of localized entities, a string (the path of a shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a

Figure 12.11: images/model_with_no_errors_and_hover.png

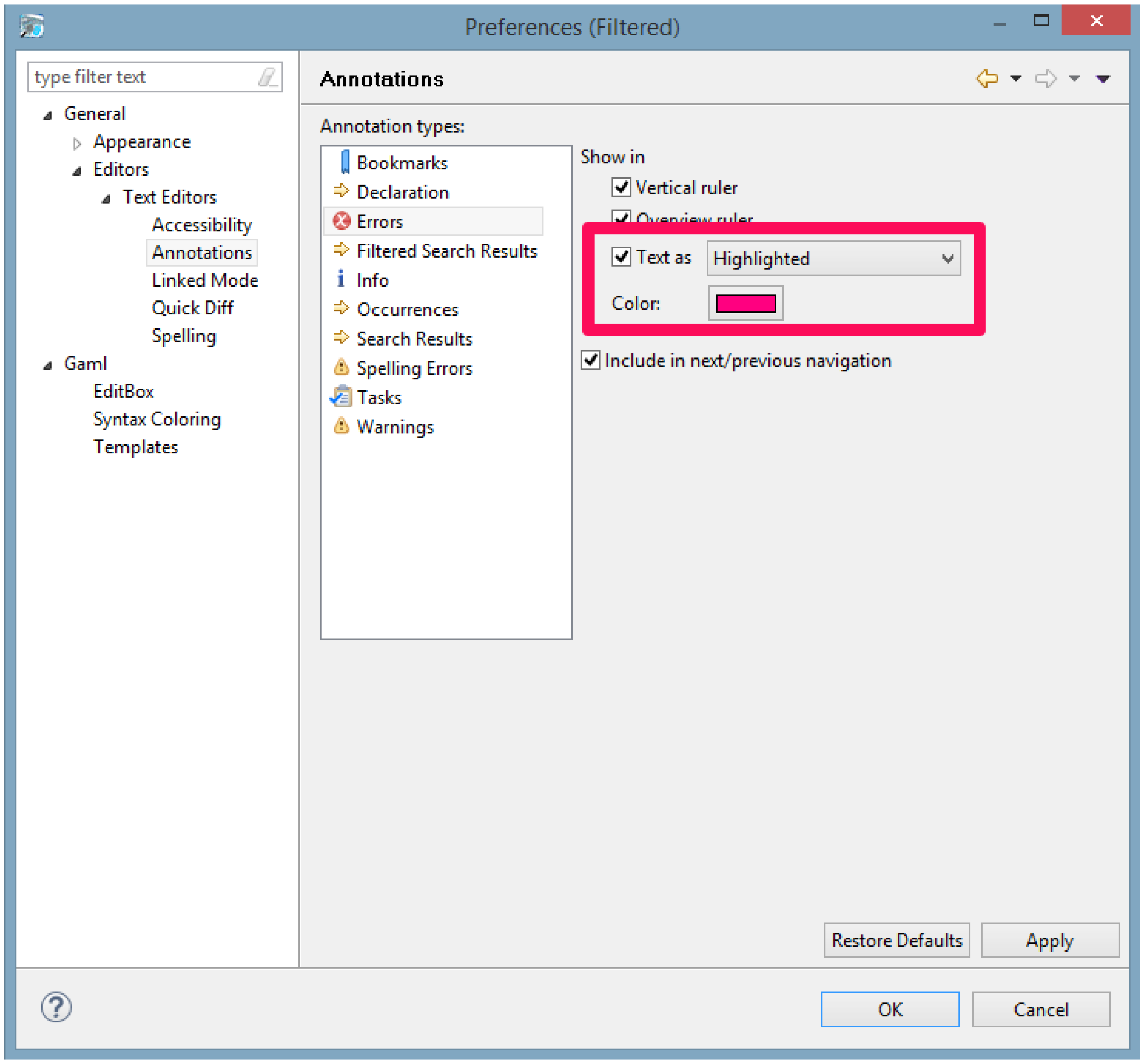
Errors in imported files

Finally, even if your model has been cleansed of all errors, it may happen that it refuses to launch because it imports another model that cannot be compiled. In the following screenshot, "My First Model.gaml" imports "My Imported Model.gaml", which sports a syntactic error.

In such a case, the importing model refuses to compile (although it is itself valid) and to propose experiments. There are cases, however, where the same importation can work. Consider the following example, where, this time, "My Imported Model.gaml" sports a semantic error in the definition of the global 'shape' attribute. Without further modifications, the use case is similar to the first one.

However, if "My First Model.gaml" happens to redefine the *shape* attribute (in global), it is now considered as valid. All the valid sections of "My Imported Model.gaml" are effectively imported, while the erroneous definition is superseded by the new one.

This process is described by the information marker next to the redefinition.


Preferences (Filtered)

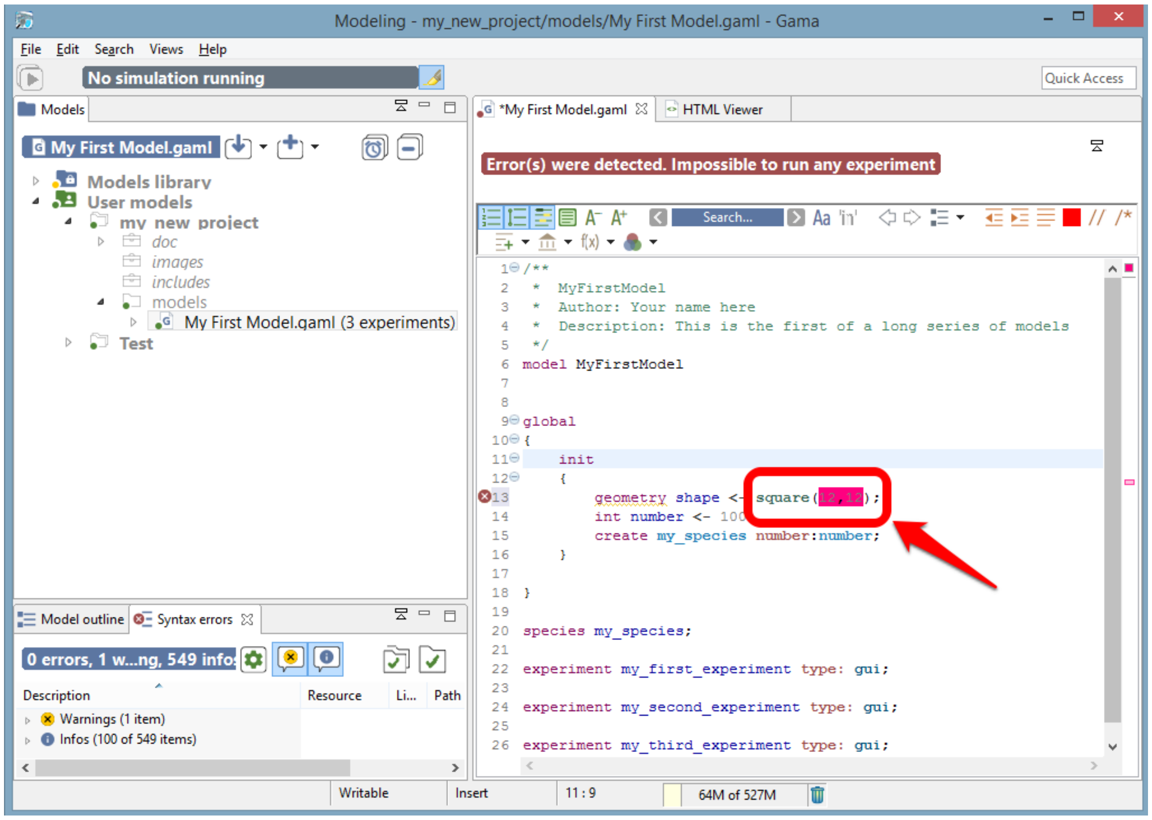
Annotations

Annotation types:
- Bookmarks
- Declaration
- Errors
- Filtered Search Results
- Info
- Occurrences
- Search Results
- Spelling Errors
- Tasks
- Warnings

Show in
- [✓] Vertical ruler
- [✓] Overview ruler
- [✓] Text as: Highlighted
- Color: 🟥

[✓] Include in next/previous navigation

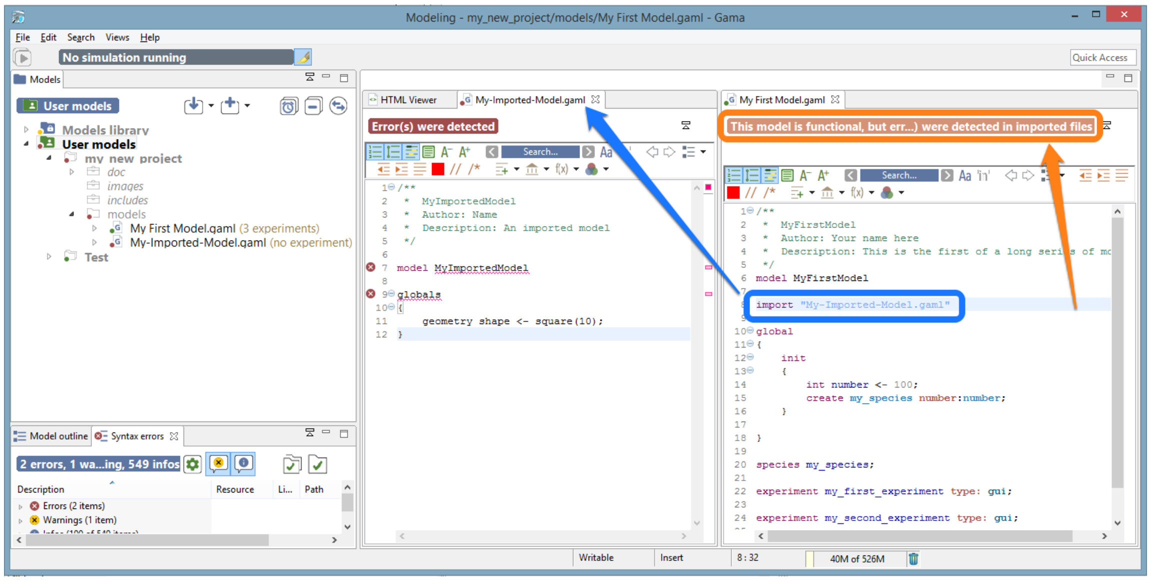
Figure 12.12: images/preferences_annotations.png



Error(s) were detected. Impossible to run any experiment

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of models
 */
model MyFirstModel


global
{
    init
    {
        geometry shape <- square(12,12);
        int number <- 100
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;

experiment my_third_experiment type: gui;
```

Figure 12.13: images/model_with_semantic_error_different_annotation.png



Modeling - my_new_project/models/My First Model.gaml - Gama

Error(s) were detected

```
/**
 *  MyImportedModel
 *  Author: Name
 *  Description: An imported model
 */

model MyImportedModel

globals
{
	geometry shape <- square(10);
}
```

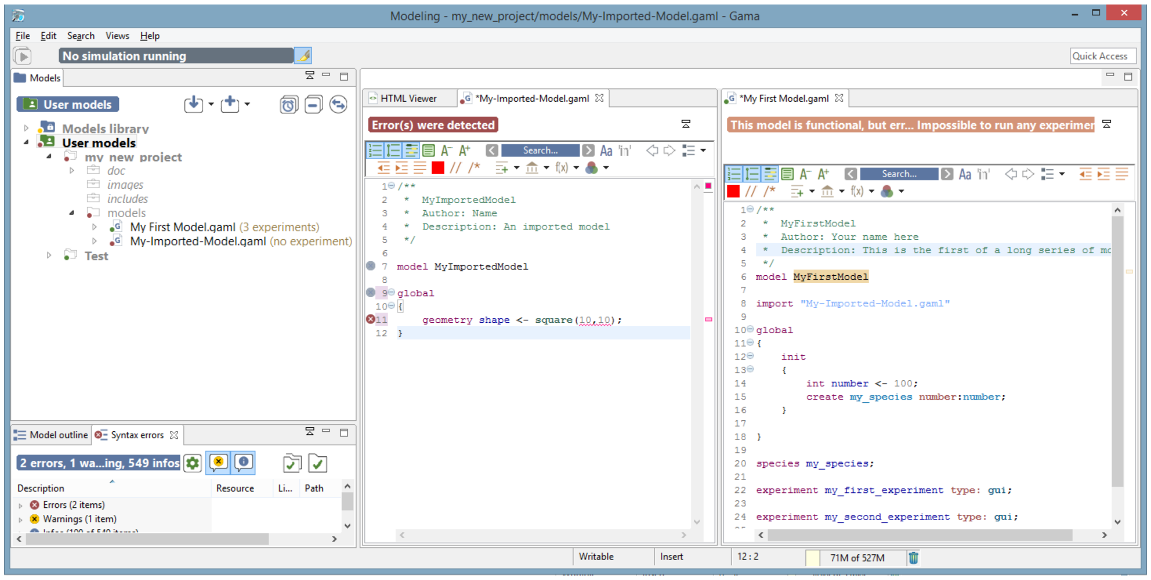
This model is functional, but err...) were detected in imported files

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of mo
 */
model MyFirstModel

import "My-Imported-Model.gaml"

global
{
	init
	{
		int number <- 100;
		create my_species number:number;
	}

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;
```

2 errors, 1 wa...ing, 549 infos

Description	Resource	Li...	Path
Errors (2 items)			
Warnings (1 item)			

Figure 12.14: images/model_with_imported_errors.png



Modeling - my_new_project/models/My-Imported-Model.gaml - Gama

Error(s) were detected

```
/**
 *  MyImportedModel
 *  Author: Name
 *  Description: An imported model
 */

model MyImportedModel

global
{
	geometry shape <- square(10,10);
}
```

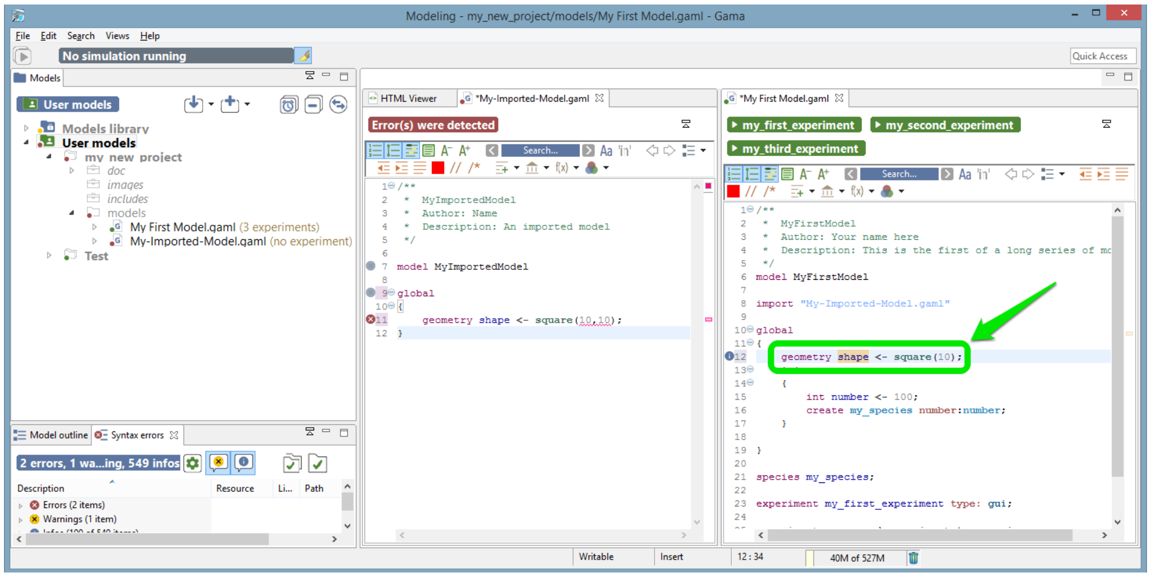
This model is functional, but err... Impossible to run any experimen

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of mo
 */
model MyFirstModel

import "My-Imported-Model.gaml"

global
{
	init
	{
		int number <- 100;
		create my_species number:number;
	}

}

species my_species;

experiment my_first_experiment type: gui;

experiment my_second_experiment type: gui;
```

2 errors, 1 wa...ing, 549 infos

Description	Resource	Li...	Path
Errors (2 items)			
Warnings (1 item)			

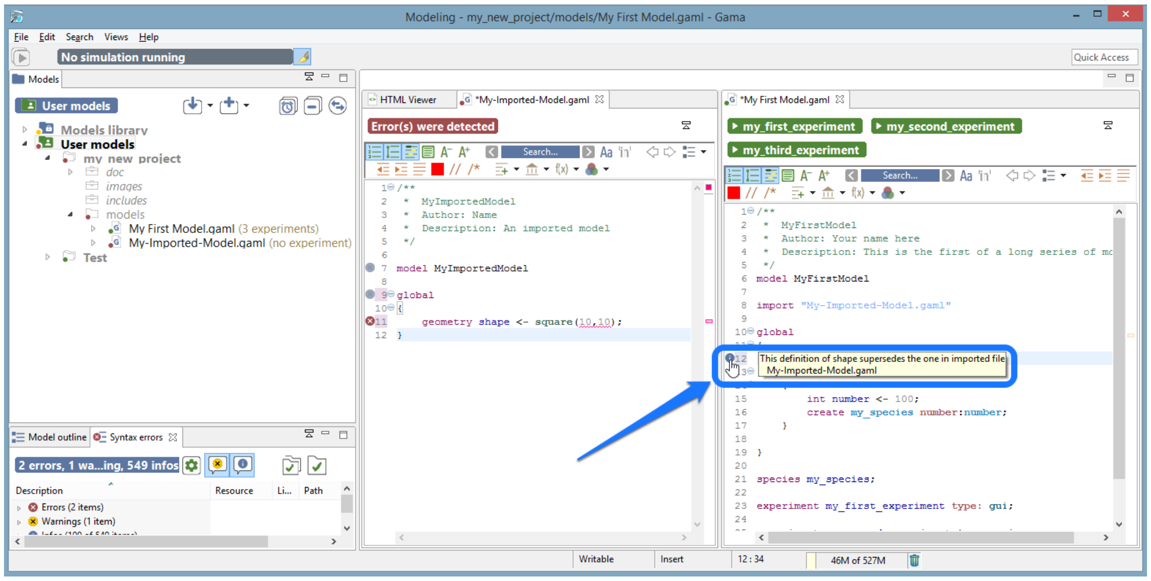
Figure 12.15: images/model_with_imported_semantic_error.png



```
/**
 *  MyImportedModel
 *  Author: Name
 *  Description: An imported model
 */

model MyImportedModel

global
{
    geometry shape <- square(10,10);
}
```

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of mo
 */
model MyFirstModel

import "My-Imported-Model.gaml"

global
{
    geometry shape <- square(10);

    {
        int number <- 100;
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

```

Figure 12.16: images/model_with_superseded_semantic_error.png



This definition of shape supersedes the one in imported file My-Imported-Model.gaml

```
/**
 *  MyFirstModel
 *  Author: Your name here
 *  Description: This is the first of a long series of mo
 */
model MyFirstModel

import "My-Imported-Model.gaml"

global

        int number <- 100;
        create my_species number:number;
    }

}

species my_species;

experiment my_first_experiment type: gui;

```

Figure 12.17: images/model_with_superseded_semantic_error_and_hover.png

Cleaning models

It may happen that the metadata that GAMA maintains about the different projects (which includes the various **markers** on files in the workspace, etc.) becomes corrupted from time to time. This especially happens if you frequently switch workspaces, but not only. In those (hopefully rare) cases, GAMA may report incorrect errors for perfectly legible files.

When such odd behaviors are detected, or if you want to regularly keep your metadata in a good shape, you can clean all your project, by clicking on the button "Clear and validate all projects" (in the syntax errors view).

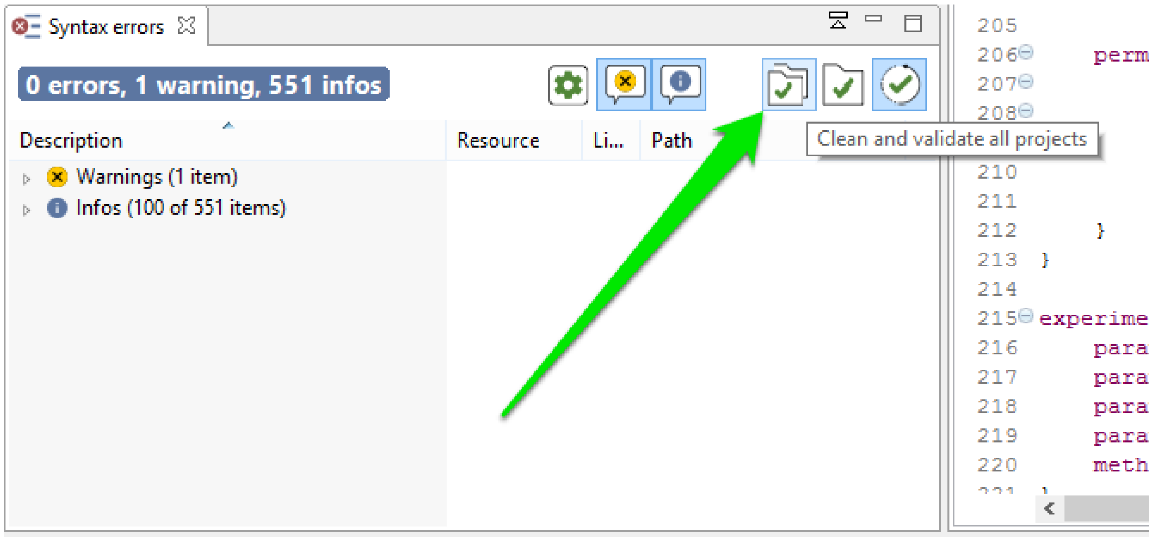


Figure 12.18: images/action_clean.png

Running Experiments

Running an experiment is the only way, in GAMA, to execute simulations on a model. Experiments can be run in different ways. 1. The first, and most common way, consists in launching an experiment from the Modeling perspective, using the user interface proposed by the simulation perspective to run simulations. 1. The second way, detailed on this page, allows to automatically launch an experiment when opening GAMA, subsequently using the same user interface. 1. The last way, known as running headless experiments, does not make use of the user interface and allows to manipulate GAMA entirely from the command line.

All three ways are strictly equivalent in terms of computations (with the exception of the last one omitting all the computations necessary to render simulations on displays or in the UI). They simply differ by their usage: 1. The first one is heavily used when designing models or demonstrating several models. 1. The second is intended to be used when demonstrating or experimenting a single model. 1. The last one is useful when running large sets of simulations, especially over networks or grids of computers.

Chapter 13

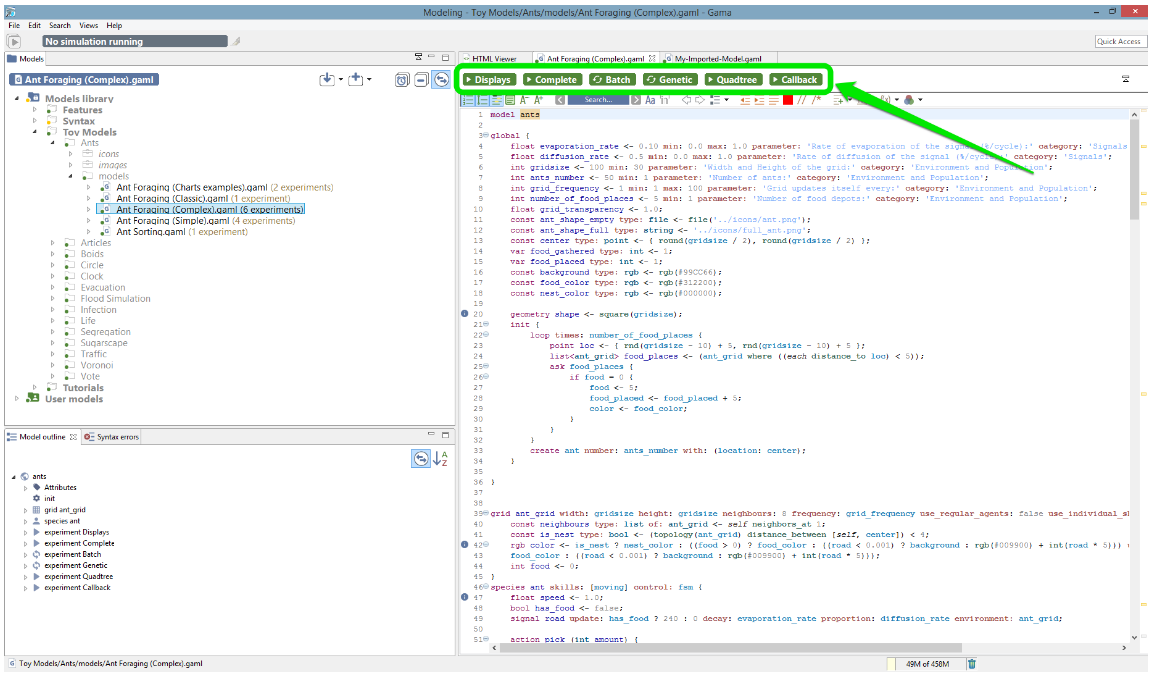
Launching Experiments from the User Interface

GAMA supports multiple ways of launching experiments from within the Modeling Perspective, in editors or in the navigator.

Table of contents

From an Editor

As already mentioned on this page, GAML editors will provide the easiest way to launch experiments. Whenever a model that contains the definition of experiments is validated, these experiments will appear as distinct buttons, in the order in which they are defined in the file, in the header ribbon above the text. Simply clicking one of these buttons launches the corresponding experiment.



```
model ants

global {
    float evaporation_rate <- 0.10 min: 0.0 max: 1.0 parameter: 'Rate of evaporation of the signal (%/cycle):' category: 'Signals
    float diffusion_rate <- 0.5 min: 0.0 max: 1.0 parameter: 'Rate of diffusion of the signal (%/cycle):' category: 'Signals';
    int gridsize <- 100 min: 30 parameter: 'Width and Height of the grid:' category: 'Environment and Population';
    int ants_number <- 50 min: 1 parameter: 'Number of ants:' category: 'Environment and Population';
    int grid_frequency <- 1 min: 1 max: 100 parameter: 'Grid updates itself every:' category: 'Environment and Population';
    int number_of_food_places <- 5 min: 1 parameter: 'Number of food depots:' category: 'Environment and Population';
    float grid_transparency <- 1.0;
    const ant_shape_empty type: file <- file('../icons/ant.png');
    const ant_shape_full type: string <- '../icons/full_ant.png';
    const center type: point <- { round(gridsize / 2), round(gridsize / 2) };
    var food_gathered type: int <- 1;
    var food_placed type: int <- 1;
    const background type: rgb <- rgb(#99CC66);
    const food_color type: rgb <- rgb(#312200);
    const nest_color type: rgb <- rgb(#000000);

    geometry shape <- square(gridsize);
    init {
        loop times: number_of_food_places {
            point loc <- { rnd(gridsize - 10) + 5, rnd(gridsize - 10) + 5 };
            list<ant_grid> food_places <- (ant_grid where ((each distance_to loc) < 5));
            ask food_places {
                if food = 0 {
                    food <- 5;
                    food_placed <- food_placed + 5;
                    color <- food_color;
                }
            }
        }
        create ant number: ants_number with: (location: center);
    }

}


grid ant_grid width: gridsize height: gridsize neighbours: 8 frequency: grid_frequency use_regular_agents: false use_individual_sl
    const neighbours type: list of: ant_grid <- self neighbors_at 1;
    const is_nest type: bool <- (topology(ant_grid) distance_between [self, center]) < 4;
    rgb color <- is_nest ? nest_color : ((food > 0) ? food_color : ((road < 0.001) ? background : rgb(#009900) + int(road * 5))) 
    food_color : ((road < 0.001) ? background : rgb(#009900) + int(road * 5)));
    int food <- 0;
}
species ant skills: [moving] control: fsm {
    float speed <- 1.0;
    bool has_food <- false;
    signal road update: has_food ? 240 : 0 decay: evaporation_rate proportion: diffusion_rate environment: ant_grid;

    action pick (int amount) {
```

Figure 13.1: images/editor_launch.png

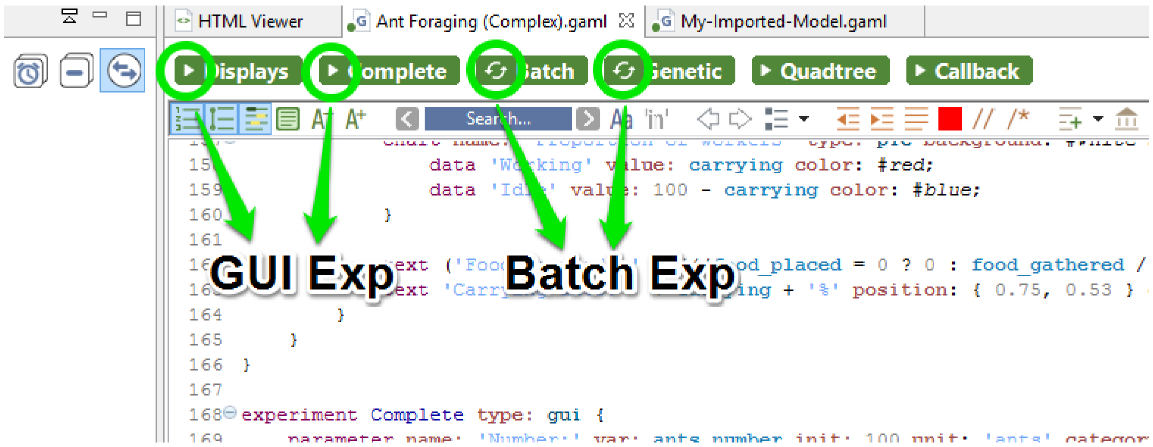
For each of those launching buttons, you can see 2 different pictograms, showing the type of experiment. An experiment can either be a GUI Experiment or a Batch Experiment.

From the Navigator

You can also launch your experiments from the navigator, by expanding a model and double clicking on one of the experiments available (The number of experiments for each model is visible also in the navigator). As for the editor, the two types of experimentations (gui and batch) are differentiated by a pictogram.

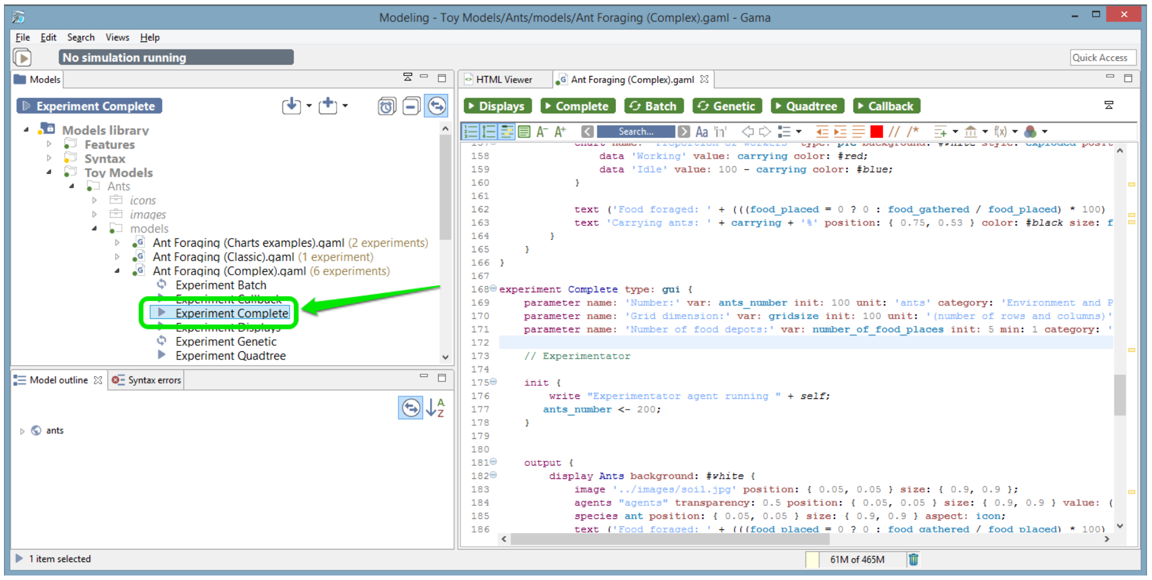
Running Experiments Automatically

Once an experiment has been launched (unless it is run in headless mode, of course), it normally displays its views and waits from an input from the user, usually a click on the "Run" or "Step" buttons (see here).



```
                data 'Working' value: carrying color: #red;
                data 'Idle' value: 100 - carrying color: #blue;
            }

            text ('Foo                 (((food_placed = 0 ? 0 : food_gathered /
            text 'Carry                 ing + '%' position: { 0.75, 0.53 }
        }
    }
}

experiment Complete type: gui {
    parameter name: 'Number:' var: ants_number init: 100 unit: 'ants' categor
```

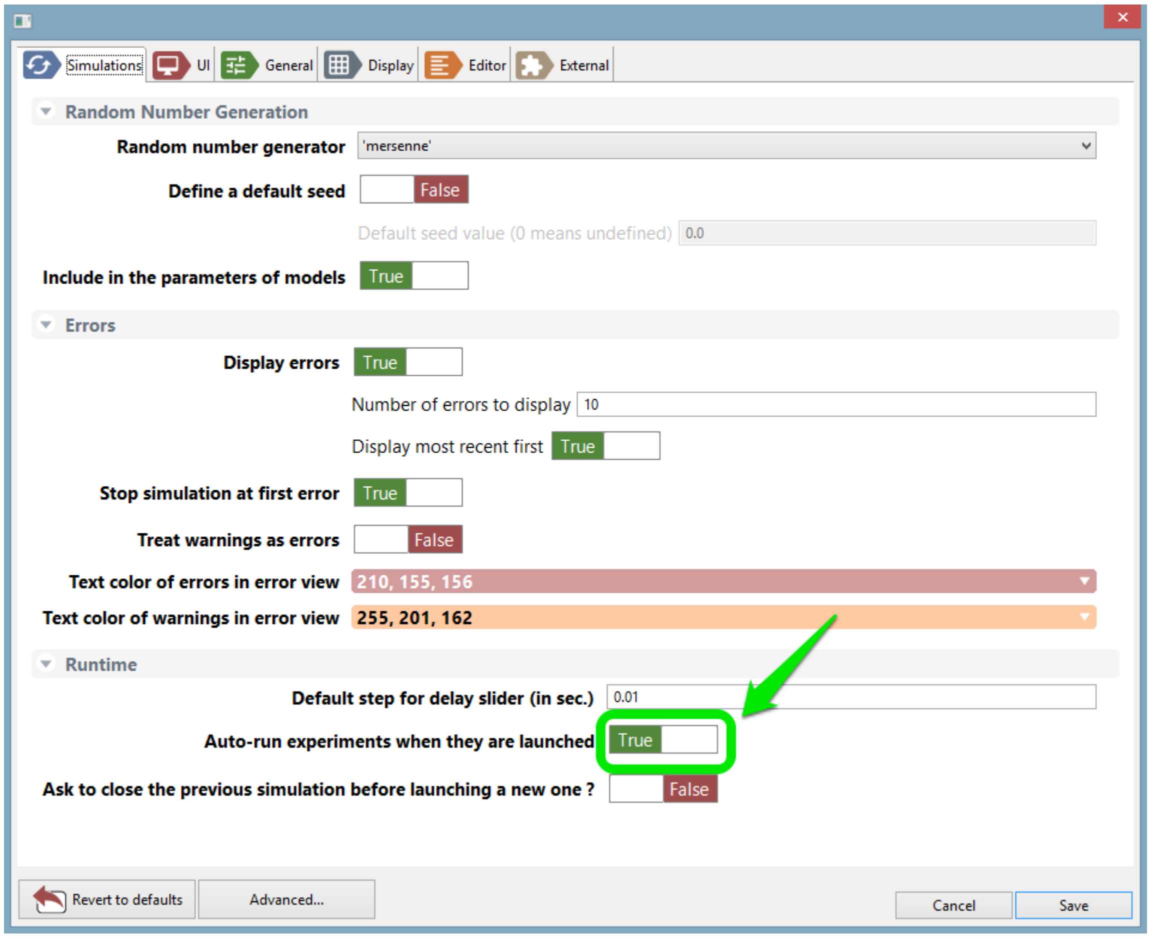
Figure 13.2: images/editor_different_types_of_experiment.png



```
                data 'Working' value: carrying color: #red;
                data 'Idle' value: 100 - carrying color: #blue;
            }

            text ('Food foraged: ' + (((food_placed = 0 ? 0 : food_gathered / food_placed) * 100)
            text 'Carrying ants: ' + carrying + '%' position: { 0.75, 0.53 } color: #black size: f
        }
    }
}

experiment Complete type: gui {
    parameter name: 'Number:' var: ants_number init: 100 unit: 'ants' category: 'Environment and P
    parameter name: 'Grid dimension:' var: gridsize init: 100 unit: '(number of rows and columns)'
    parameter name: 'Number of food depots:' var: number_of_food_places init: 5 min: 1 category: '

    // Experimentator

    init {
        write "Experimentator agent running " + self;
        ants_number <- 200;
    }


    output {
        display Ants background: #white {
            image '../images/soil.jpg' position: { 0.05, 0.05 } size: { 0.9, 0.9 };
            agents "agents" transparency: 0.5 position: { 0.05, 0.05 } size: { 0.9, 0.9 } value: (
            species ant position: { 0.05, 0.05 } size: { 0.9, 0.9 } aspect: icon;
            text ('Food foraged: ' + (((food_placed = 0 ? 0 : food_gathered / food_placed) * 100)
```

Figure 13.3: images/navigator_launch.png

It is however possible to make experiments run directly once launched, without requiring any intervention from the user. To install this feature, open the preferences of GAMA. On the first tab, simply check "Auto-run experiments when they are launched" (which is unchecked by default) and hit "OK" to dismiss the dialog. Next time you'll launch an experiment, it will run automatically (this option also applies to experiments launched from the command line).



Random Number Generation

Random number generator 'mersenne'

Define a default seed False

Default seed value (0 means undefined) 0.0

Include in the parameters of models True

Errors

Display errors True

Number of errors to display 10

Display most recent first True

Stop simulation at first error True

Treat warnings as errors False

Text color of errors in error view 210, 155, 156

Text color of warnings in error view 255, 201, 162

Runtime

Default step for delay slider (in sec.) 0.01

Auto-run experiments when they are launched True

Ask to close the previous simulation before launching a new one ? False

Figure 13.4: images/prefs_auto_run.png

Running Several Simulations

It is possible in GAMA to run several simulations. Each simulation will be launched with the same seed (which means that if the parameters are the same, then the result will be exactly the same). All those simulations are synchronized in the same cycle.

To run several experiments, you have to write it directly in your model.

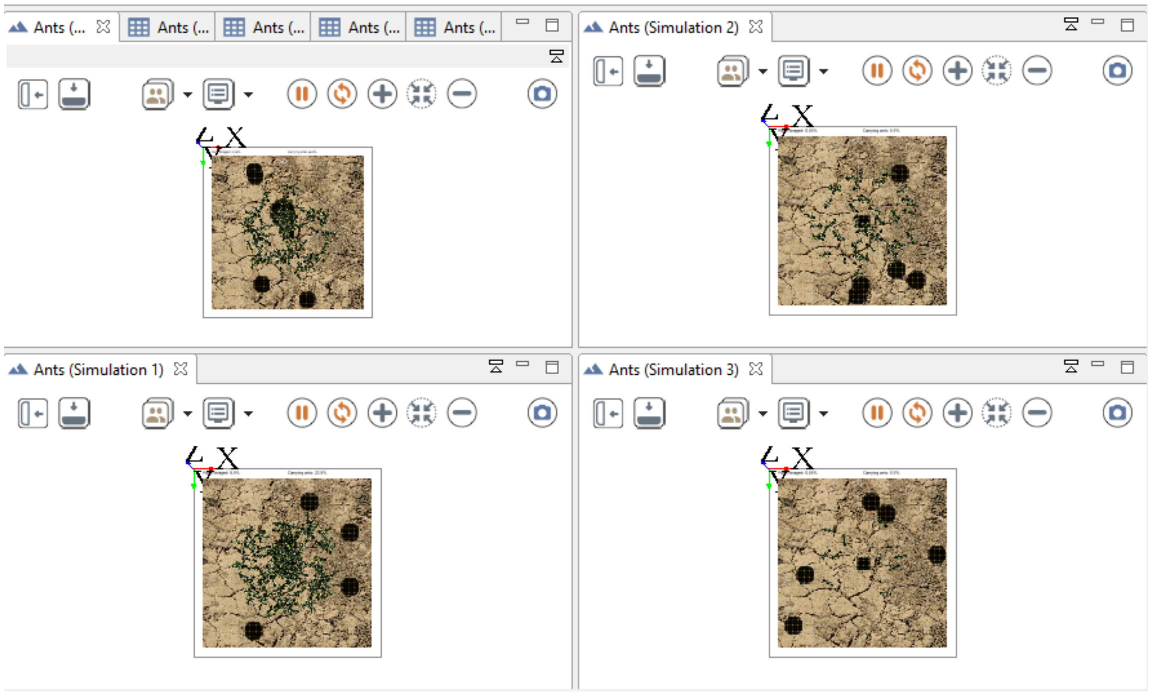


Figure 13.5: images/run_several_simulations.png

Chapter 14

Experiments User Interface

As soon as an experiment is launched, the modeler is facing a new environment (with different menus and views) called the *Simulation Perspective*). The *Navigator* is still opened in this perspective, though, and it is still possible to edit models in it, but it is considered as good practice to use each perspective for what is has been designed for. Switching perspectives is easy. The small button in the top-left corner of the window allows to switch back and forth the two perspectives.

The actual contents of the simulation perspective will depend on the experiment being run and the outputs it defines. The next sections will present the most common ones (inspectors, monitors and displays), as well as the views that are not defined in outputs, like the Parameters or Errors view. An overview of the menus and commands specific to the simulation perspective is also available.

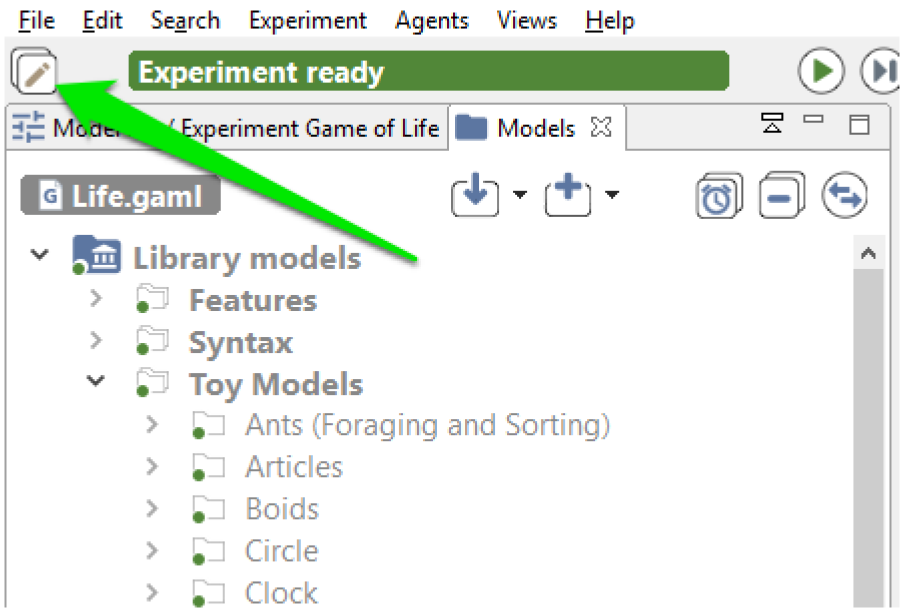


Figure 14.1: images/button_switch.png

Chapter 15

Menus and Commands

The simulation perspective adds on the user interface a number of new menus and commands (i.e. buttons) that are specific to experiment-related tasks.

Table of contents

Experiment Menu

A menu, called "Experiment", allows to control the current experiment. It shares some of its commands with the general toolbar (see below).

- **Run/Pause**: allows to run or pause the experiment depending on its current state.
- **Step by Step**: runs the experiment for one cycle and pauses it after.
- **Reload**: stops the current experiment, deletes its contents, and reloads it, **taking into account the parameters values that might have been changed by the user**.

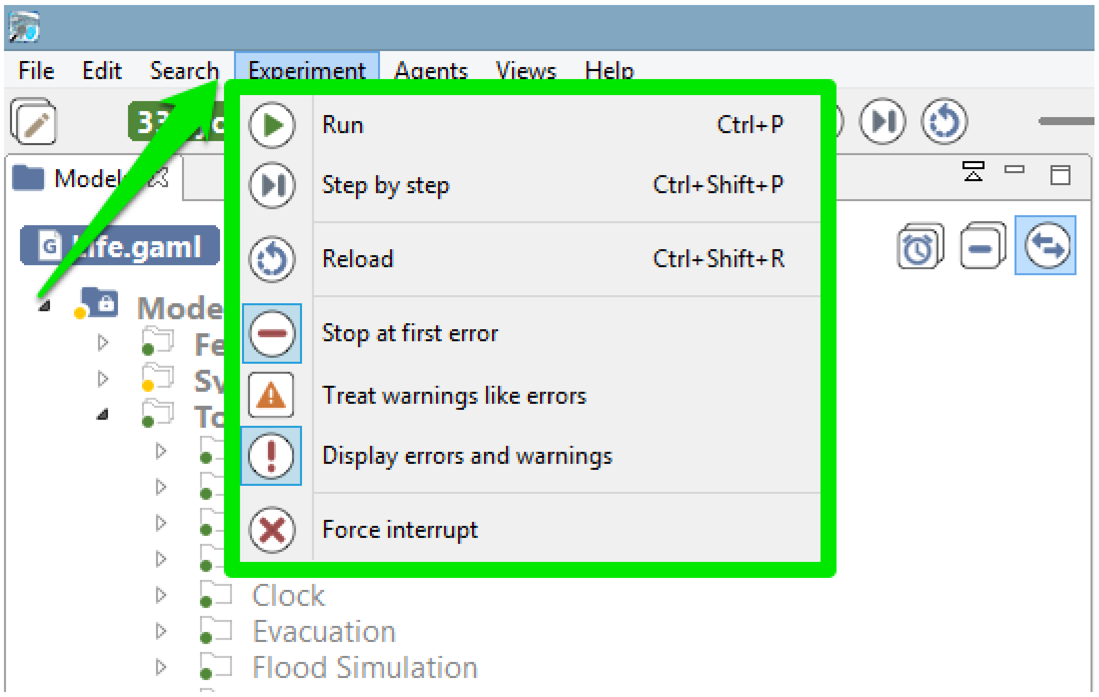


Figure 15.1: images/menu_experiment.png

- **Stop at first error**: if checked, the current experiment will stop running when an error is issued. The default value can be configured in the preferences.
- **Treat warnings as errors**: if checked, a warning will be considered as an error (and if the previous item is checked, will stop the experiment). The default value can be configured in the preferences.
- **Display warnings and errors**: if checked, displays the errors and warnings issued by the experiment. If not, do not display them. The default value can be configured in the preferences.
- **Force interrupt**: forces the experiment to stop, whatever it is currently doing, purges the memory from it, and switches to the modeling perspective. **Use this command with caution**, as it can have undesirable effects depending on the state of the experiment (for example, if it is reading files, or outputting data, etc.).

Agents Menu

A second menu is added in the simulation perspective: "Agents". This menu allows for an easy access to the different agents that populate an experiment.

This hierarchical menu is always organized in the same way, whatever the experiment being run. A first level is dedicated to the current simulation agent: it allows to browse its population or to inspect the simulation agent itself. Browsing the population will give access to the current experiment agent (the "host" of this population). A second level lists the "micro-populations" present in the simulation agent. And the third level will give access to each individual agent in these populations. This organization is of course recursive: if these agents are themselves hosts of micro-populations, they will be displayed in their individual menu.

Each agent, when selected, will reveal a similar individual menu. This menu will contain a set of predefined actions, the commands defined by the user for this species, if any, and then the micro-populations hosted by this agent, if any. Agents (like the instances of "ant" below) that do not host other agents and whose species has no user commands will have a "simple" individual menu.

These are the 4 actions that will be there most of the time:

- **Inspect**: open an inspector on this agent.
- **Highlight**: makes this agent the current "highlighted" agent, forcing it to appear "highlighted" in all the displays that might have been defined.

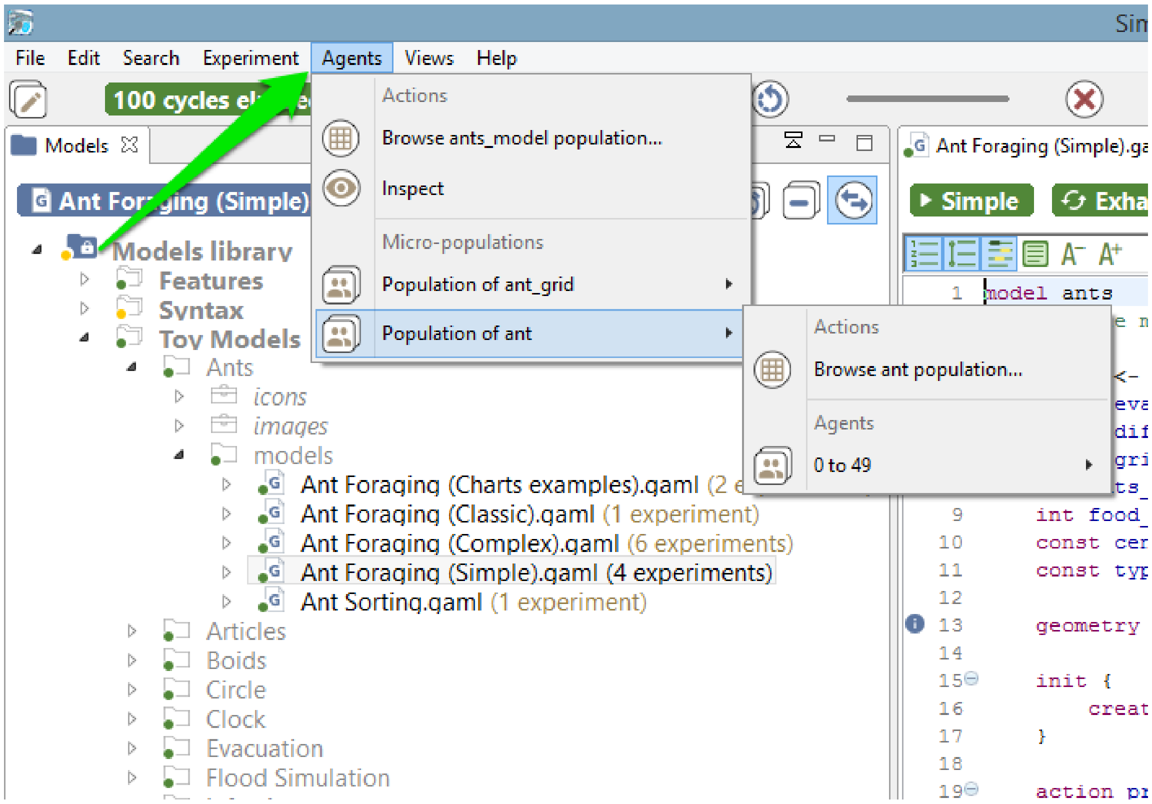


Figure 15.2: images/menu_agents.png

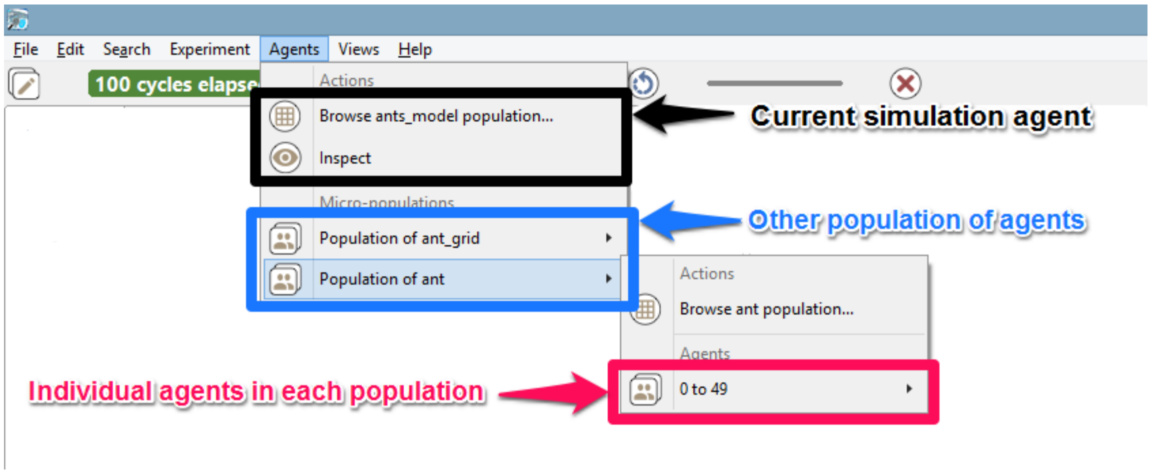


Figure 15.3: images/menu_agents_2.png

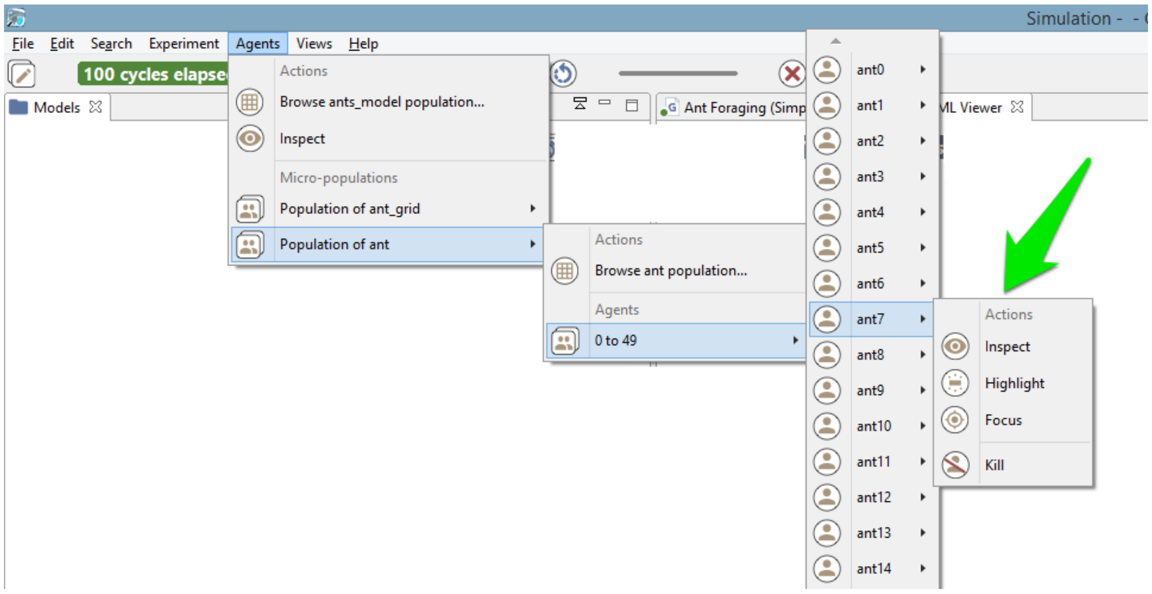


Figure 15.4: images/menu_agents_3.png

- **Focus**: this option is not accessible if no displays are defined. Makes the current display zoom on the selected agent (if it is displayed) so that it occupies the whole view.
- **Kill**: destroys the selected agent and disposes of it. **Use this command with caution**, as it can have undesirable effects if the agent is currently executing its behavior.

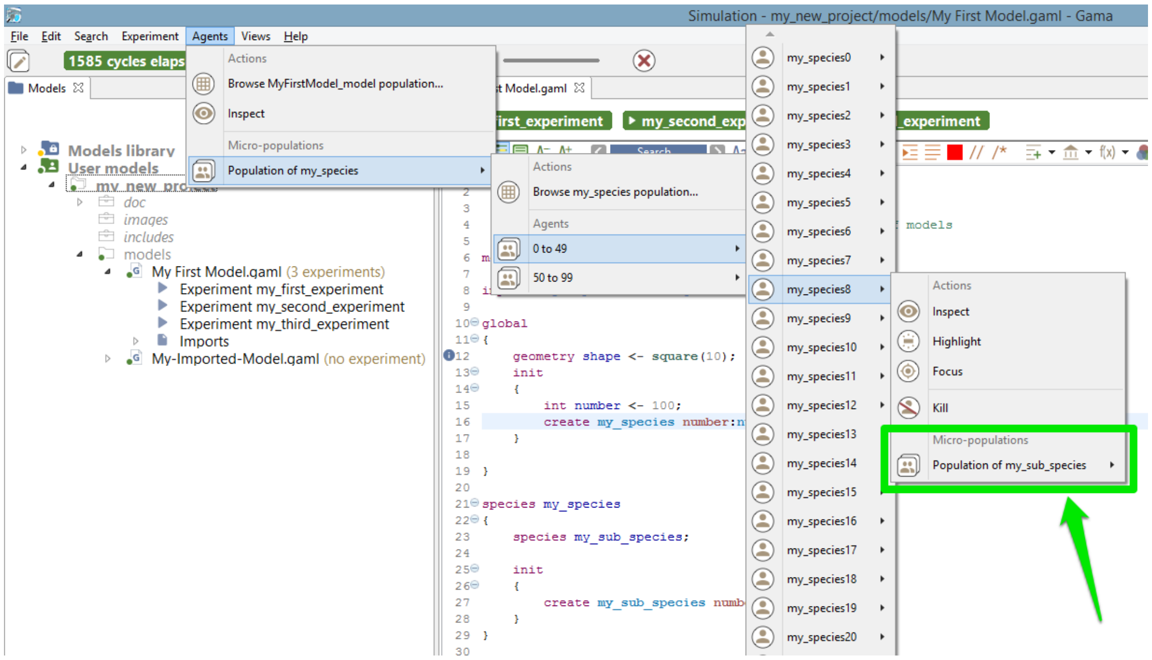
If an agent hosts other agents (it is the case in multi-level architecture), you can access to the micro-population quite easily:

If user commands are defined for a species (for example in the existing model Features/Driving Skill/Road Traffic simple (City)), their individual menu will look like the following:

General Toolbar

The last piece of user interface specific to the Simulation Perspective is a toolbar, which contains controls and information displays related to the current experiment.

This toolbar is voluntarily minimalist, with three buttons already present in the experiment menu (namely, “Play/Pause”, “Step by Step” and “Reload”), which don’t need to be detailed



```
global
{
	geometry shape <- square(10);
	init
	{
		int number <- 100;
		create my_species number:n
	}

}

species my_species
{
	species my_sub_species;

	init
	{
		create my_sub_species numb
	}
}
```

Figure 15.5: images/menu_agents_multi_level.png

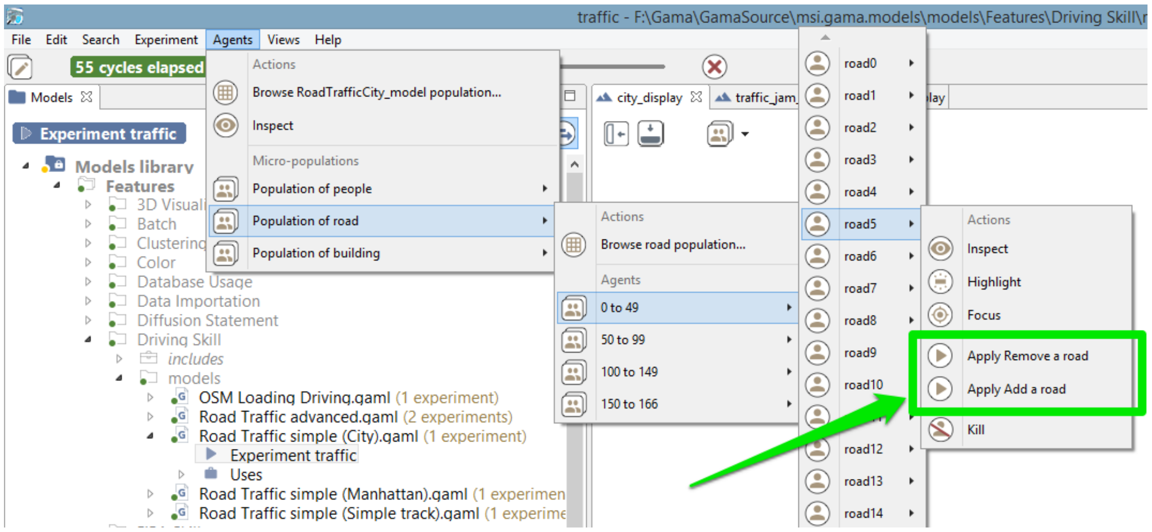


Figure 15.6: images/menu_agents_user_command.png

here, and two new controls ("Experiment status" and "Cycle Delay"), which are explained below.

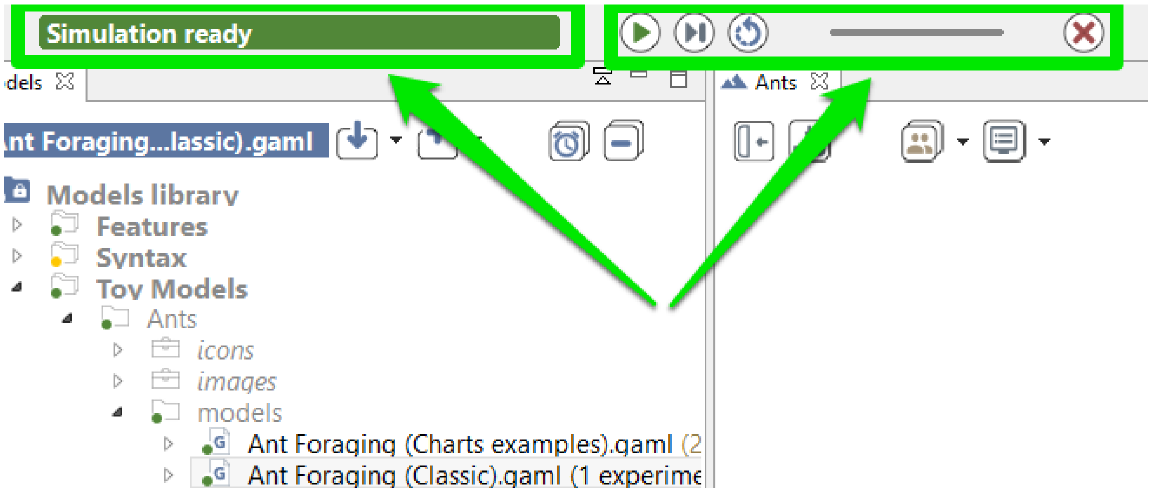

Simulation ready
dels
Ants
nt Foraging...lassic).gaml
Models library
Features
Syntax
Toy Models
Ants
icons
images
models
Ant Foraging (Charts examples).gaml (2
Ant Foraging (Classic).gaml (1 experime

Figure 15.7: images/toolbar.png

While opening an experiment, the status will display some information about what's going on. For instance, that GAMA is busy instantiating the agents, or opening the displays.

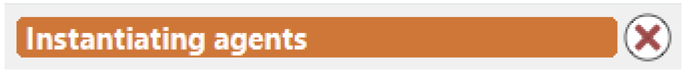


Figure 15.8: images/toolbar_instantiating_agents.png

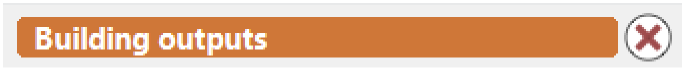


Figure 15.9: images/toolbar_building_outputs.png

The orange color usually means that, although the experiment is not ready, things are progressing without problems (a red color message is an indication that something went wrong). When the loading of the experiment is finished, GAMA displays the message "Simulation ready" on a green background. If the user runs the simulation, the status changes and displays the number of cycles already elapsed in the simulation currently managed by the experiment.

Hovering over the status produces a more accurate information about the internal clock of the simulation.

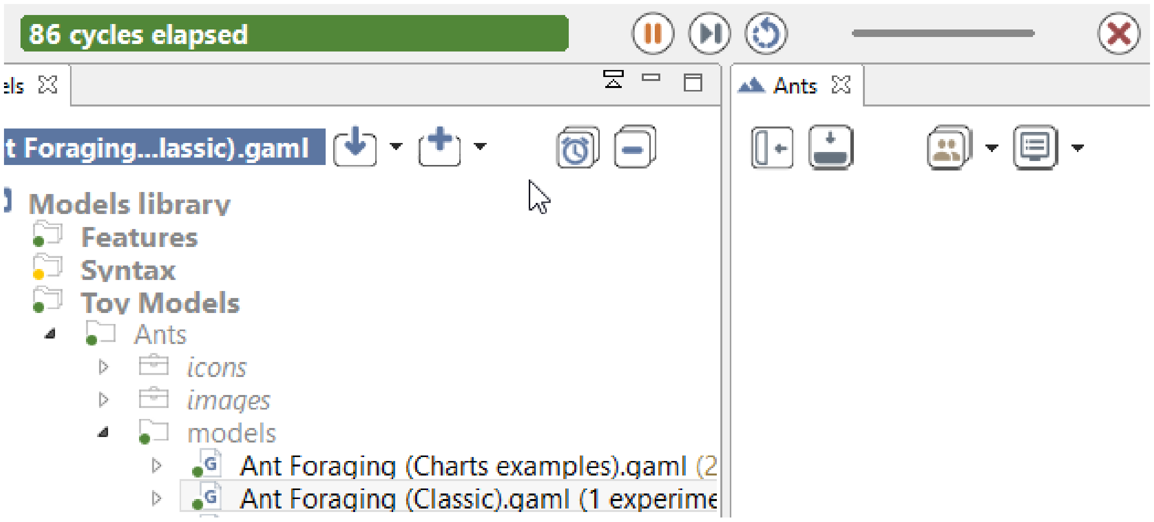
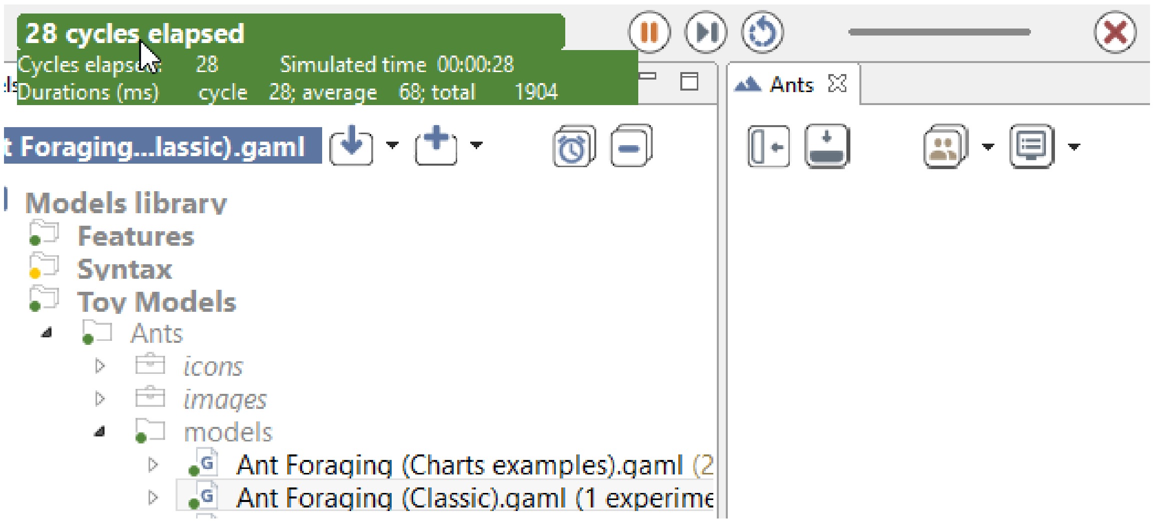

86 cycles elapsed
els
Ants
t Foraging...lassic).gaml
Models library
Features
Syntax
Toy Models
Ants
icons
images
models
Ant Foraging (Charts examples).gaml (2
Ant Foraging (Classic).gaml (1 experime

Figure 15.10: images/toolbar_running.png


28 cycles elapsed
Ants
t Foraging...lassic).gaml
Models library
Features
Syntax
Toy Models
Ants
icons
images
models
Ant Foraging (Charts examples).gaml (2
Ant Foraging (Classic).gaml (1 experime

```
Cycles elapsed:   28       Simulated time  00:00:28
Durations (ms)    cycle   28; average   68; total     1904
```

Figure 15.11: images/toolbar_running_with_info.png

From top to bottom of this hover, we find the number of cycles elapsed, the simulated time already elapsed (in the example above, one cycle lasts one second of *simulated time*), the duration of cycle in milliseconds, the average duration of one cycle (computed over the number of cycles elapsed), and the total duration, so far, of the simulation (still in milliseconds).

Although these durations are entirely dependent on the speed of the simulation engine (and, of course, the number of agents, their behaviors, etc.), there is a way to control it partially with the second control, which allows the user to force a minimal duration (in milliseconds) for a cycle, from 0 (its initial position) to 1000. Note that this minimal duration (or delay) will remain the same for the subsequent reloads of the experiment.

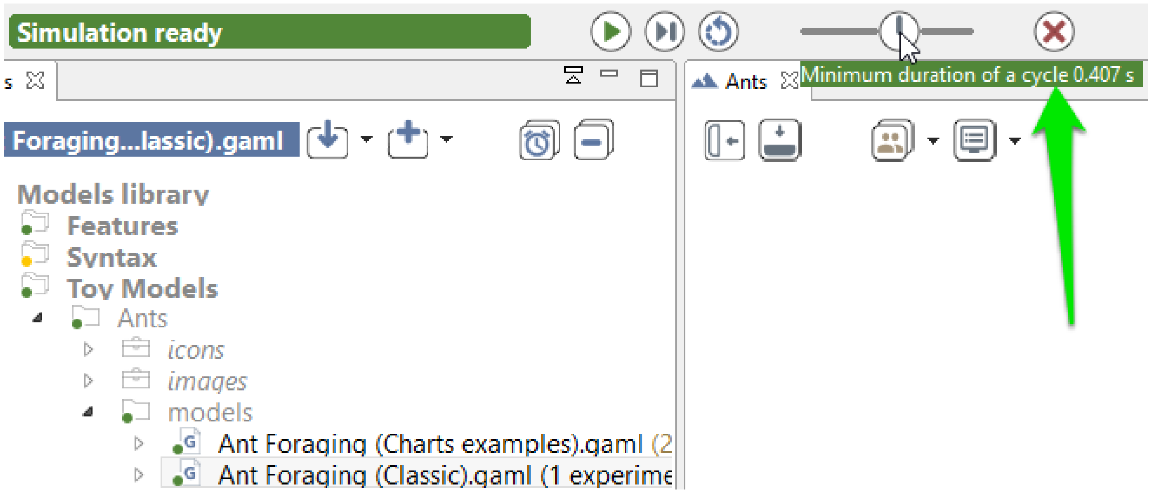

Simulation ready
Minimum duration of a cycle 0.407 s
Ants
Foraging...lassic).gaml
Models library
Features
Syntax
Toy Models
Ants
icons
images
models
Ant Foraging (Charts examples).gaml (2
Ant Foraging (Classic).gaml (1 experime

Figure 15.12: images/toolbar_running_with_delay.png

In case it is necessary to have more than 1s of delay, it has to be defined, instead, as an attribute of the experiment.

Chapter 16

Parameters View

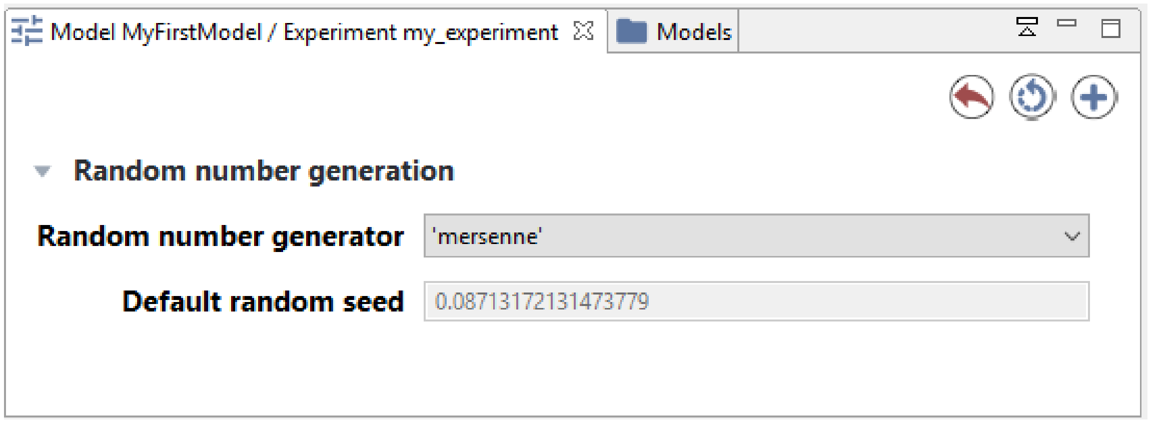
In the case of an experiment, the modeler can define the parameters he wants to be able to modify to explore the simulation, and thus the ones he wants to be able to display and alter in the GUI interface.

It important to notice that all modification made in the parameters are used for simulation reload only. Creation of a new simulation from the model will erase the modifications.

Table of contents

Built-in parameters

Every GUI experiment displays a pane named "Parameters" containing at least two built-in parameters related to the random generator: * the Random Number Generator, with a choice between 3 RNG implementations, * the Random Seed


Model MyFirstModel / Experiment my_experiment

Random number generation

Random number generator 'mersenne'

Default random seed 0.08713172131473779

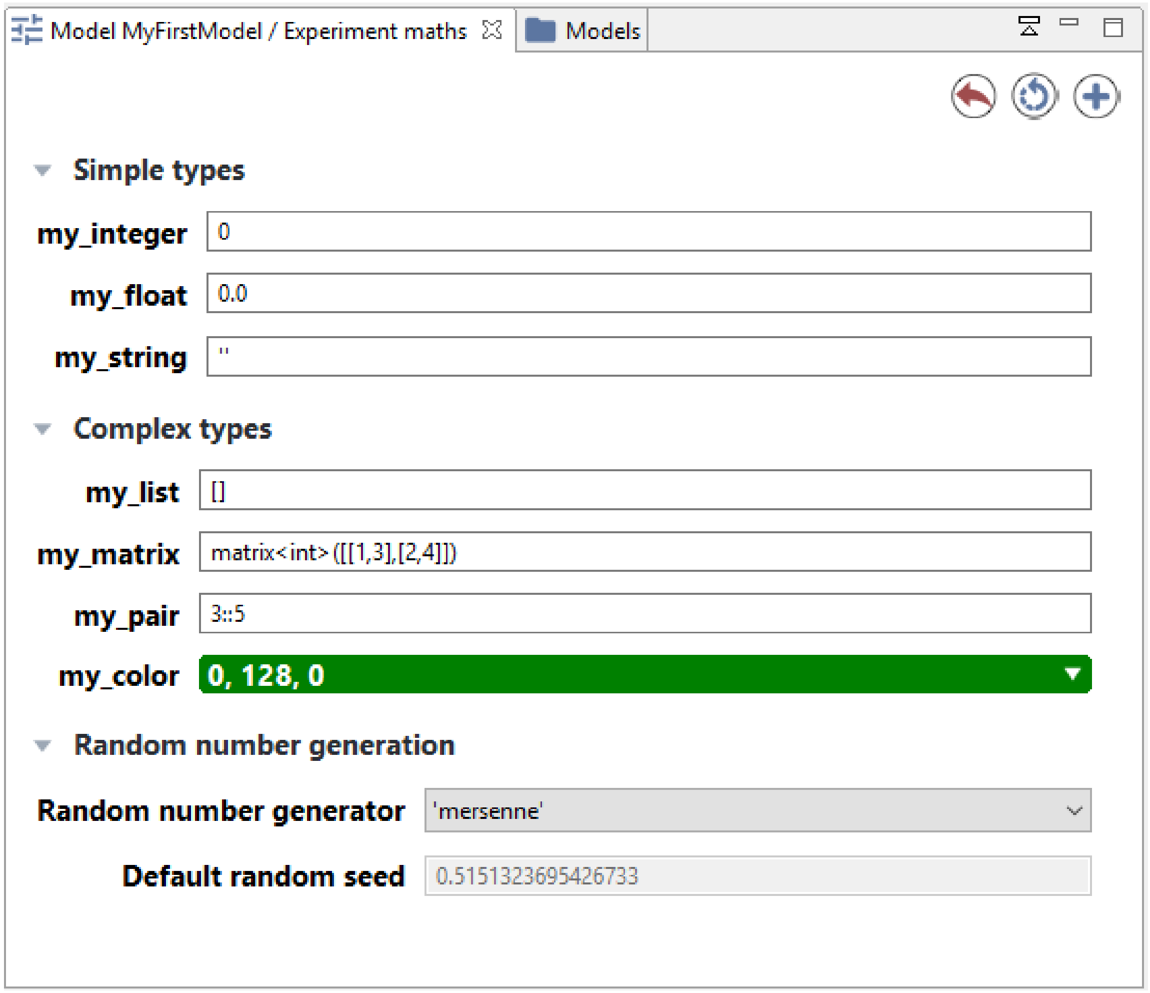
Figure 16.1: images/parameters_built_in.png

Parameters View

The modeler can define himself parameters that can be displayed in the GUI and that are sorted by categories. Note that the interface will depend on the data type of the parameter: for example, for integer or float parameters, a simple text box will be displayed whereas a color selector will be available for color parameters. The parameters value displayed are the initial value provided to the variables associated to the parameters in the model.

The above parameters view is generated from the following code:

```
global
{
    int i;
    float f;
    string s;
    list l;
    matrix m;
    pair p;
    rgb c;
}

experiment maths type: gui {
    parameter "my_integer" var: i <- 0 category:"Simple types";
    parameter "my_float" var: f <- 0.0 category:"Simple types";
    parameter "my_string" var: s <- "" category:"Simple types";
```



Simple types

my_integer 0

my_float 0.0

my_string ''

Complex types

my_list []

my_matrix matrix<int>([[1,3],[2,4]])

my_pair 3::5

my_color 0, 128, 0

Random number generation

Random number generator 'mersenne'

Default random seed 0.5151323695426733

Figure 16.2: images/parameters.png

```
    parameter "my_list" var: l <- [] category:"Complex types";
    parameter "my_matrix" var: m <- matrix([[1,2],[3,4]])
   category:"Complex types";
    parameter "my_pair" var: p <- 3::5 category:"Complex types";
    parameter "my_color" var: c <- #green category:"Complex types
   ";

    output {}
}
```

Click on Edit button in case of list or map parameters or the color or matrix will open an additional window to modify the parameter value.

Modification of parameters values

The modeler can modify the parameter values. After modifying the parameter values, you can reload the simulation by clicking on the top-right circular arrow button.

You can also add a new simulation to the old one, using those new parameters, by clicking on the top-right plus symbol button.

If he wants to come back to the initial value of parameters, he can click on the top-right red curved arrow of the parameters view.

Chapter 17

Inspectors and monitors

GAMA offers some tools to obtain informations about one or several agents. There are two kinds of tools : * agent browser * agent inspector

GAMA offers as well a tool to get the value of a specific expression: monitors.

Table of contents

Agent Browser

The species browser provides informations about all or a selection of agents of a species.

The agent browser is available through the **Agents** menu or by right clicking on a display (screenshots from the).

It displays in a table all the values of the agent variables of the considered species; each line corresponding to an agent. The list of attributes is displayed on the left side of the view, and you can select the attributes you want to be displayed, simply by clicking on it (Ctrl + Click for multi-selection).

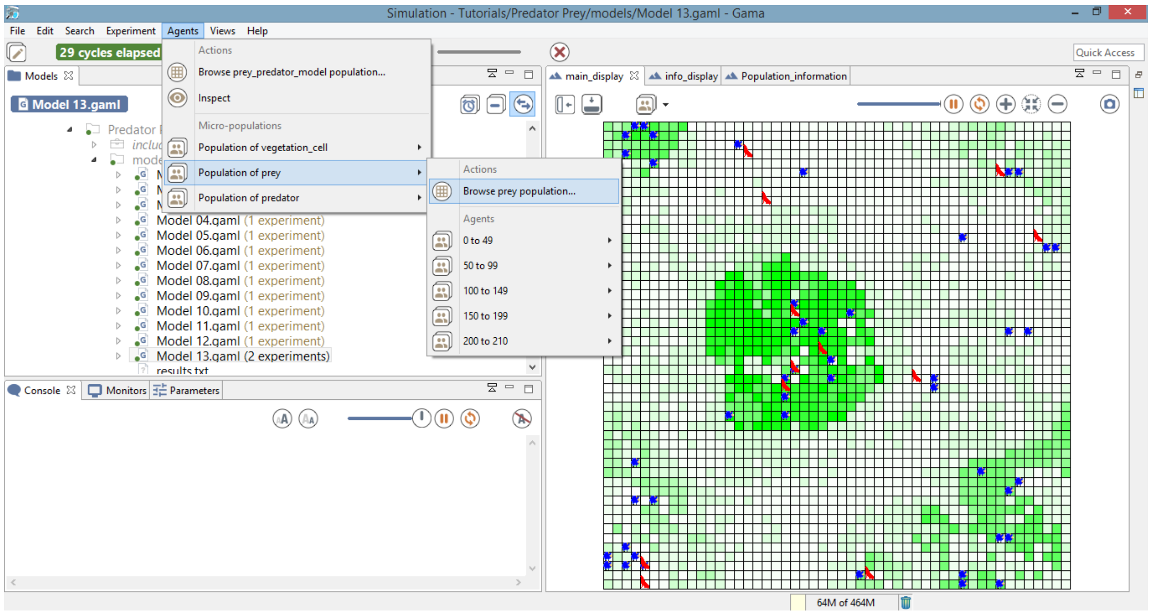

Simulation - Tutorials/Predator Prey/models/Model 13.gaml - Gama
File Edit Search Experiment Agents Views Help
29 cycles elapsed
Actions
Browse prey_predator_model population...
Inspect
Micro-populations
Population of vegetation_cell
Population of prey
Population of predator
Actions
Browse prey population...
Agents
0 to 49
50 to 99
100 to 149
150 to 199
200 to 210
Models
Model 13.gaml
Model 04.gaml (1 experiment)
Model 05.gaml (1 experiment)
Model 06.gaml (1 experiment)
Model 07.gaml (1 experiment)
Model 08.gaml (1 experiment)
Model 09.gaml (1 experiment)
Model 10.gaml (1 experiment)
Model 11.gaml (1 experiment)
Model 12.gaml (1 experiment)
Model 13.gaml (2 experiments)
results.txt
Console Monitors Parameters
main_display info_display Population_information
Quick Access
64M of 464M

Figure 17.1: images/browse-menu.png

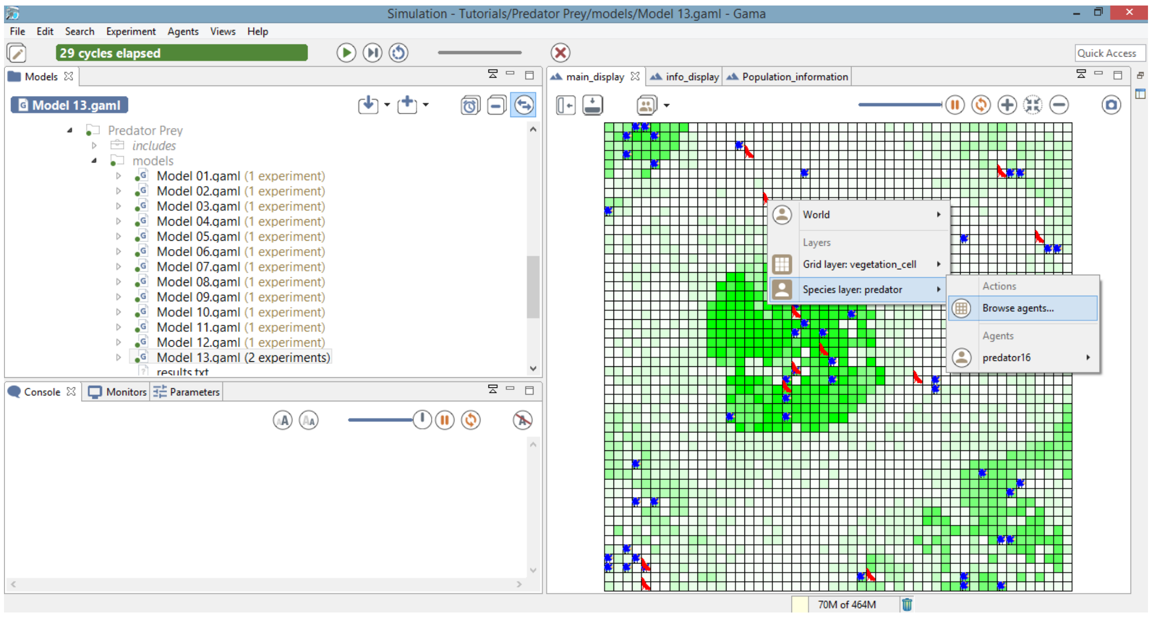
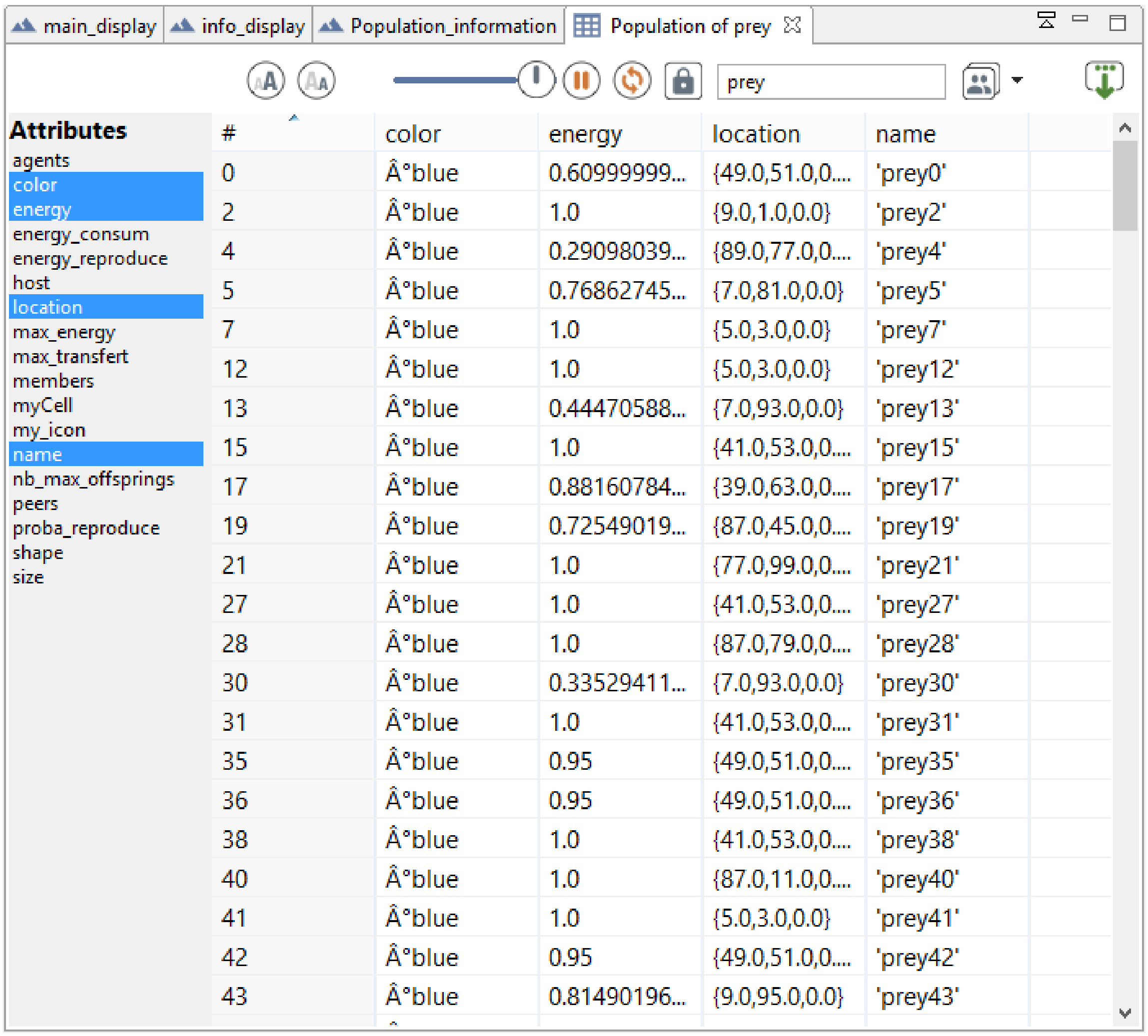

Simulation - Tutorials/Predator Prey/models/Model 13.gaml - Gama
File Edit Search Experiment Agents Views Help
29 cycles elapsed
Models
Model 13.gaml
Predator Prey
includes
models
Model 01.gaml (1 experiment)
Model 02.gaml (1 experiment)
Model 03.gaml (1 experiment)
Model 04.gaml (1 experiment)
Model 05.gaml (1 experiment)
Model 06.gaml (1 experiment)
Model 07.gaml (1 experiment)
Model 08.gaml (1 experiment)
Model 09.gaml (1 experiment)
Model 10.gaml (1 experiment)
Model 11.gaml (1 experiment)
Model 12.gaml (1 experiment)
Model 13.gaml (2 experiments)
results.txt
Console Monitors Parameters
main_display info_display Population_information
Quick Access
World
Layers
Grid layer: vegetation_cell
Species layer: predator
Actions
Browse agents...
Agents
predator16
70M of 464M

Figure 17.2: images/browse_right_clicking.png



Attributes

- agents
- color
- energy
- energy_consum
- energy_reproduce
- host
- location
- max_energy
- max_transfert
- members
- myCell
- my_icon
- name
- nb_max_offsprings
- peers
- proba_reproduce
- shape
- size

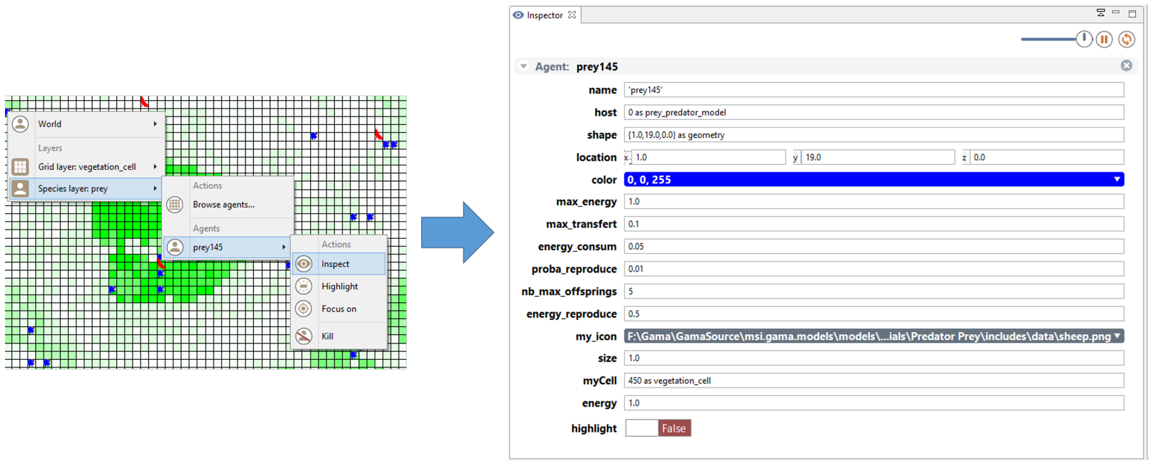
#	color	energy	location	name
0	Â°blue	0.60999999...	{49.0,51.0,0....	'prey0'
2	Â°blue	1.0	{9.0,1.0,0.0}	'prey2'
4	Â°blue	0.29098039...	{89.0,77.0,0....	'prey4'
5	Â°blue	0.76862745...	{7.0,81.0,0.0}	'prey5'
7	Â°blue	1.0	{5.0,3.0,0.0}	'prey7'
12	Â°blue	1.0	{5.0,3.0,0.0}	'prey12'
13	Â°blue	0.44470588...	{7.0,93.0,0.0}	'prey13'
15	Â°blue	1.0	{41.0,53.0,0....	'prey15'
17	Â°blue	0.88160784...	{39.0,63.0,0....	'prey17'
19	Â°blue	0.72549019...	{87.0,45.0,0....	'prey19'
21	Â°blue	1.0	{77.0,99.0,0....	'prey21'
27	Â°blue	1.0	{41.0,53.0,0....	'prey27'
28	Â°blue	1.0	{87.0,79.0,0....	'prey28'
30	Â°blue	0.33529411...	{7.0,93.0,0.0}	'prey30'
31	Â°blue	1.0	{41.0,53.0,0....	'prey31'
35	Â°blue	0.95	{49.0,51.0,0....	'prey35'
36	Â°blue	0.95	{49.0,51.0,0....	'prey36'
38	Â°blue	1.0	{41.0,53.0,0....	'prey38'
40	Â°blue	1.0	{87.0,11.0,0....	'prey40'
41	Â°blue	1.0	{5.0,3.0,0.0}	'prey41'
42	Â°blue	0.95	{49.0,51.0,0....	'prey42'
43	Â°blue	0.81490196...	{9.0,95.0,0.0}	'prey43'

Figure 17.3: images/browse_result.png

By clicking on the right mouse button on a line, it is possible to do some action for the corresponding agent.

Agent Inspector

The agent inspector provides information about one specific agent. It also allows to change the values of its variables during the simulation. The agent inspector is available from the **Agents** menu, by right_clicking on a display, in the species inspector or when inspecting another agent.



Agent: prey145

name	'prey145'
host	0 as prey_predator_model
shape	{1.0,19.0,0.0} as geometry
location	x 1.0 y 19.0 z 0.0
color	0, 0, 255
max_energy	1.0
max_transfert	0.1
energy_consum	0.05
proba_reproduce	0.01
nb_max_offsprings	5
energy_reproduce	0.5
my_icon	F:\Gama\GamaSource\msi.gama.models\models\...ials\Predator Prey\includes\data\sheep.png
size	1.0
myCell	450 as vegetation_cell
energy	1.0
highlight	False

Figure 17.4: images/Agent_inspector.png

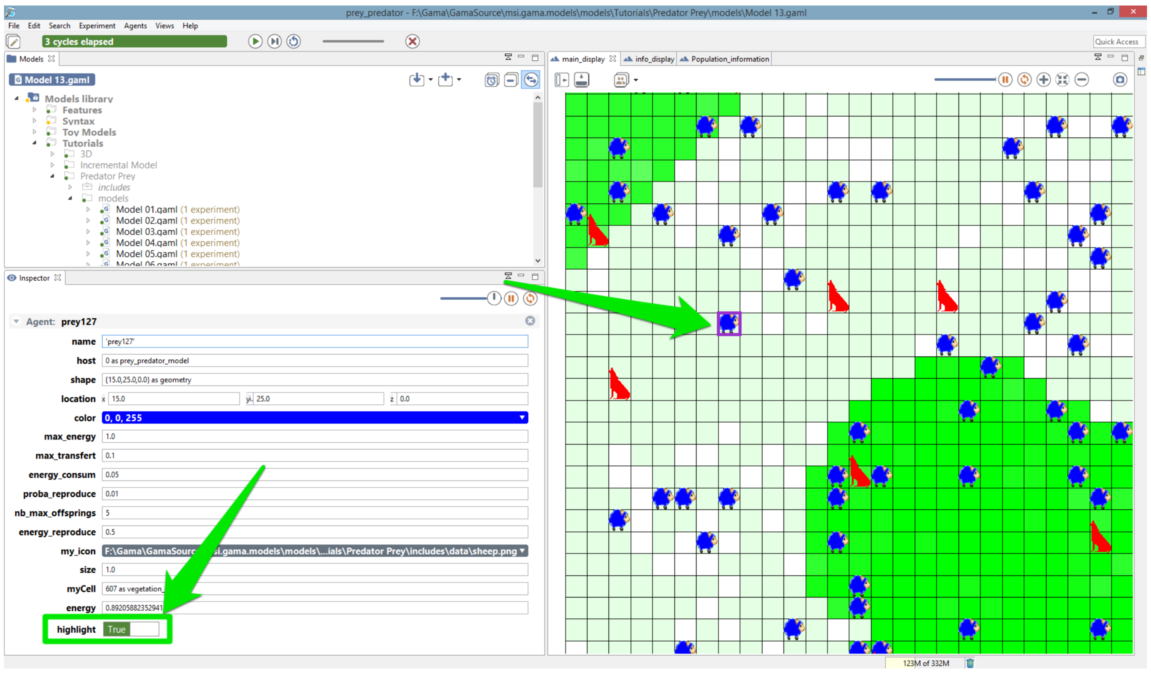
It is possible to «highlight» the selected agent.

To change the color of the highlighted agent, go to Preferences/Display.

Monitor

Monitors allow to follow the value of a GAML expression. For instance the following monitor allow to follow the number of infected people agents during the simulation. The monitor is updated at each simulation step.

It is possible to define a monitor inside a model (see this page). It is also possible to define a monitor through the graphical interface.


prey_predator - F:\Gama\GamaSource\msi.gama.models\models\Tutorials\Predator Prey\models\Model 13.gaml
File Edit Search Experiment Agents Views Help
3 cycles elapsed
Models
Model 13.gaml
Models library
Features
Syntax
Toy Models
Tutorials
3D
Incremental Model
Predator Prey
includes
models
Model 01.gaml (1 experiment)
Model 02.gaml (1 experiment)
Model 03.gaml (1 experiment)
Model 04.gaml (1 experiment)
Model 05.gaml (1 experiment)
Inspector
main_display
info_display
Population_information
Quick Access
123M of 332M

Agent: prey127

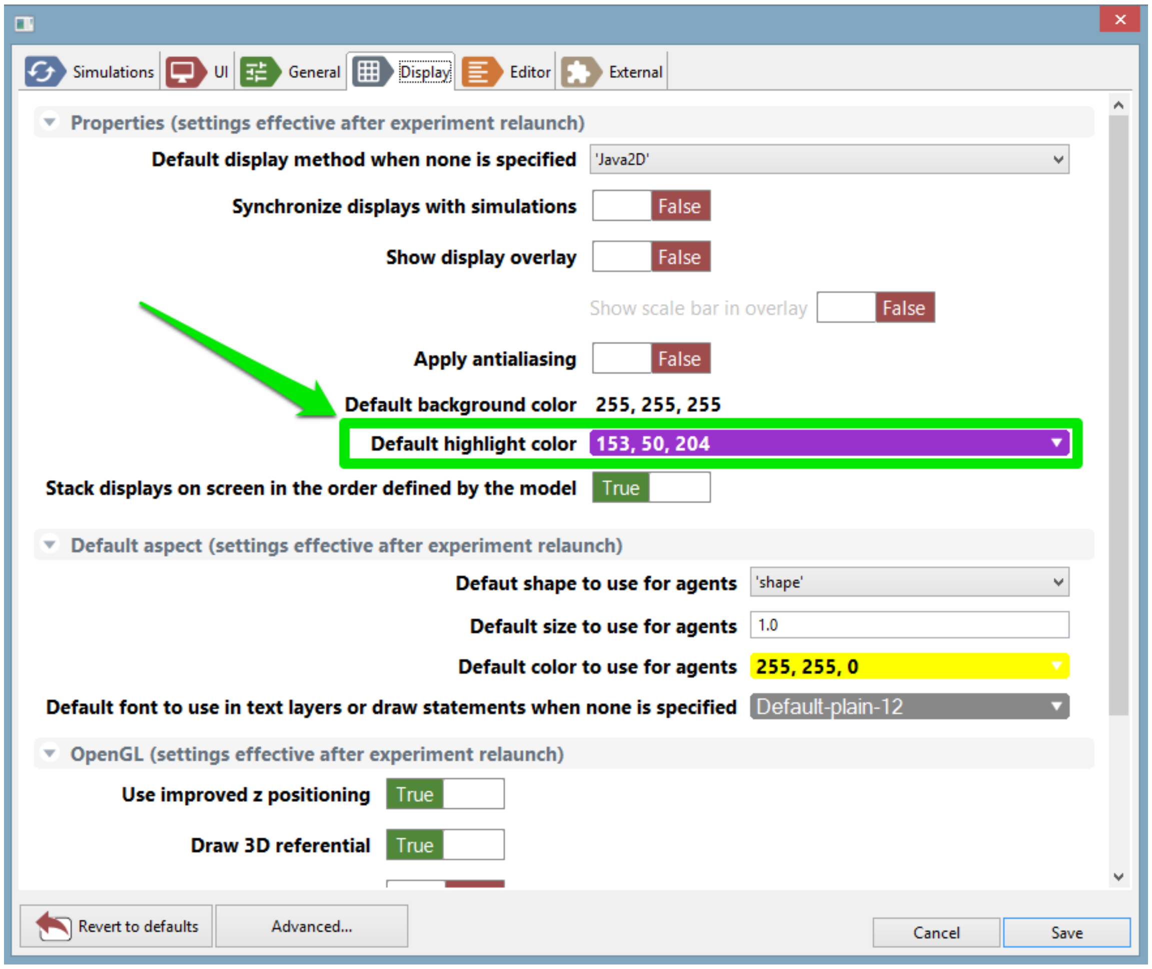
name	'prey127'
host	0 as prey_predator_model
shape	{15.0,25.0,0.0} as geometry
location	x 15.0 y 25.0 z 0.0
color	0, 0, 255
max_energy	1.0
max_transfert	0.1
energy_consum	0.05
proba_reproduce	0.01
nb_max_offsprings	5
energy_reproduce	0.5
my_icon	F:\Gama\GamaSourc[illegible]si.gama.models\models\...ials\Predator Prey\includes\data\sheep.png
size	1.0
myCell	607 as vegetation_
energy	0.89205882352941
highlight	True

Figure 17.5: images/Inspector_highlight.png

To define a monitor, first choose **Add Monitor** in the **Views** menu (or by clicking on the icon in the Monitor view), then define the display legend and the expression to monitor.

In the following example, we defined a monitor with the legend "Number initial of preys" and that has for value the global variable "nb_preys_init".

The expression should be written with the GAML language. See this page for more details about the GAML language.



Simulations UI General Display Editor External

Properties (settings effective after experiment relaunch)

Default display method when none is specified	'Java2D'
Synchronize displays with simulations	False
Show display overlay	False
Show scale bar in overlay	False
Apply antialiasing	False
Default background color	255, 255, 255
Default highlight color	153, 50, 204
Stack displays on screen in the order defined by the model	True

Default aspect (settings effective after experiment relaunch)

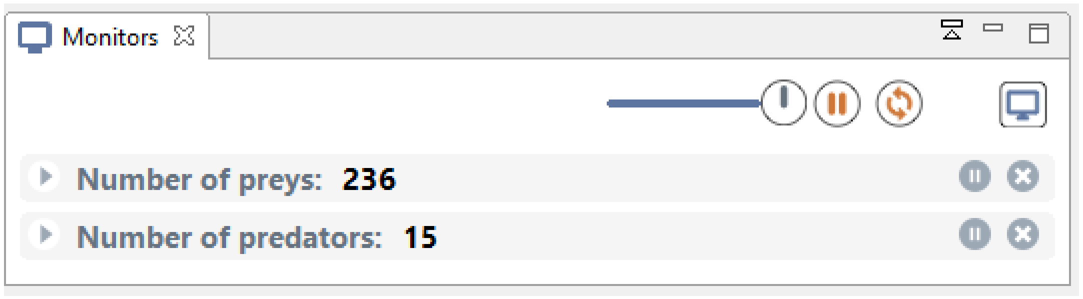
Defaut shape to use for agents	'shape'
Default size to use for agents	1.0
Default color to use for agents	255, 255, 0
Default font to use in text layers or draw statements when none is specified	Default-plain-12

OpenGL (settings effective after experiment relaunch)

Use improved z positioning	True
Draw 3D referential	True

Revert to defaults Advanced... Cancel Save

Figure 17.6: images/Inspector_change_highlight_color.png



Monitors

Number of preys: 236

Number of predators: 15

Figure 17.7: images/monitor.png

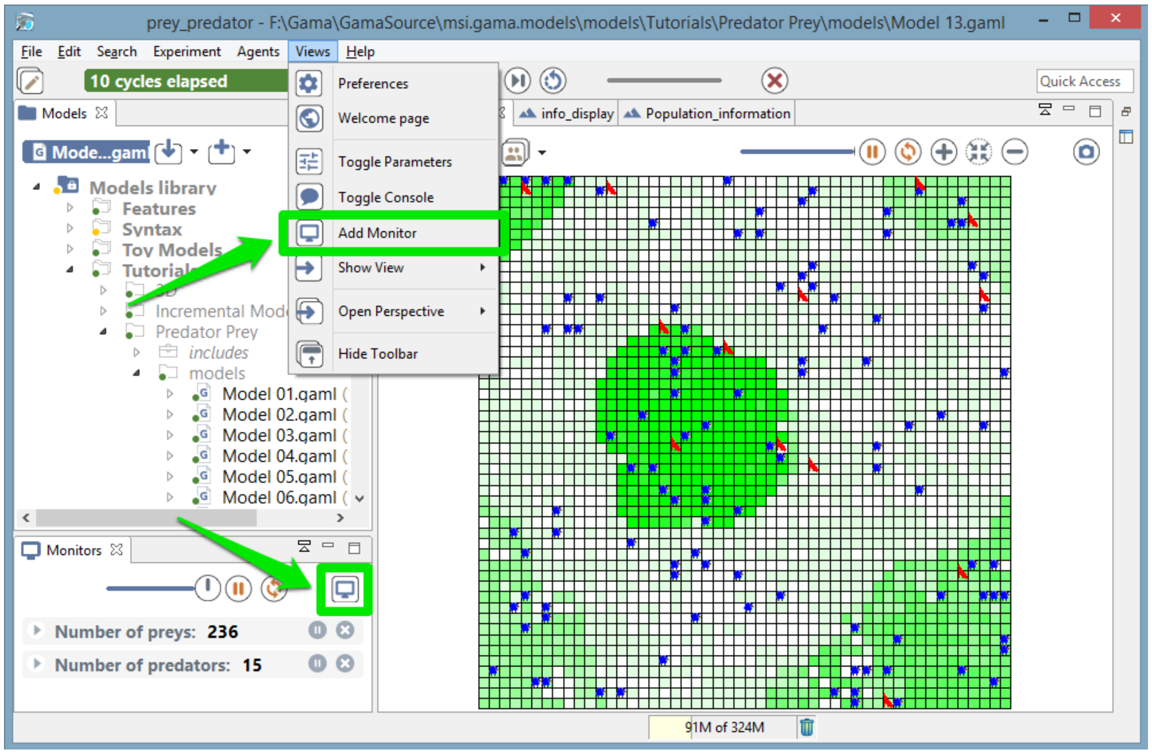
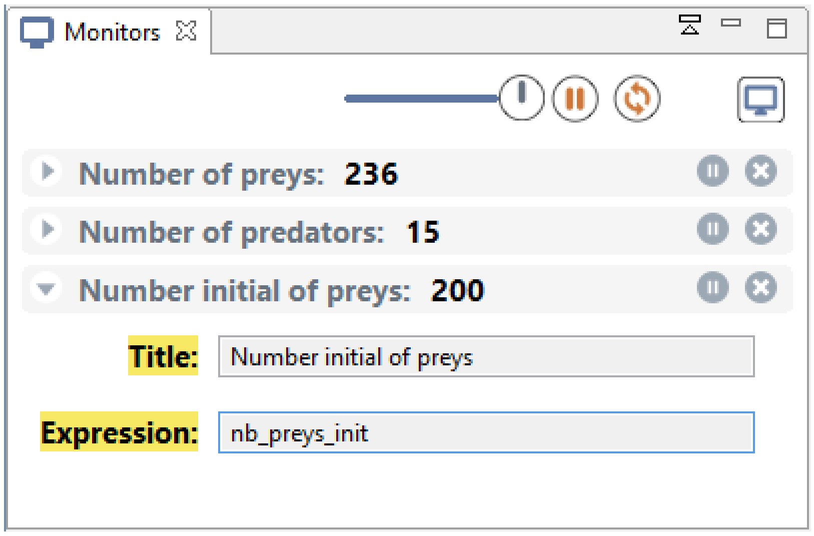


Figure 17.8: images/add_monitor.png



Number of preys: 236

Number of predators: 15

Number initial of preys: 200

Title: Number initial of preys

Expression: nb_preys_init

Figure 17.9: images/monitor_definition.png

Chapter 18

Displays

GAMA allows modelers to define several and several kinds of displays in a GUI experiment: * java 2D displays * OpenGL displays

These 2 kinds of display allows the modeler to display the same objects (agents, charts, texts ...). The OpenGL display offers extended features in particular in terms of 3D visualisation. The OpenGL displays offers in addition better performance when zooming in and out.

Table of contents

Classical displays (java2D)

The classical displays displaying any kind of content can be manipulated via the mouse (if no mouse event has been defined): * the **mouse left** press and move allows to move the camera (in 2D), * the **mouse right** click opens a context menu allowing the modeler to inspect displayed agents, * the **wheel** allows the modeler to zoom in or out.

Each display provides several buttons to manipulate the display (from left to right): * **Show/hide side bar**, * **Show/hide overlay**, * **Browse through all displayed agents**:

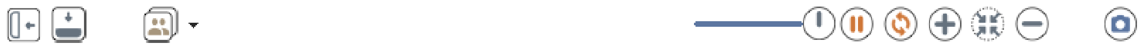


Figure 18.1: images/display-java2D.png

open a context menu to inspect agents, * **Update every X step**: configure the refresh frequence of the display, * **Pause the-display**: when pressed, the display will not be displayed anymore, the simulation is still running, * **Synchronize the display and the execution of the model**, * **Zoom in**, * **Zoom to fit view**, * **Zoom out**, * **Take a snapshot**: take a snapshot saved as a png image in the `snapshots` folder of the models folder.

The Show/Hide side bar button opens a side panel in the display allowing the modeler to configure: * **Properties** of the display: background and highlight color, display the scale bar * For each layer, we can configure visibility, transparency, position and size of the layer. For grid layers, we can in addition show/hide grids. For species layers, we can also configure the displayed aspect. For text layers, we can the expression displayed with the color and the font.

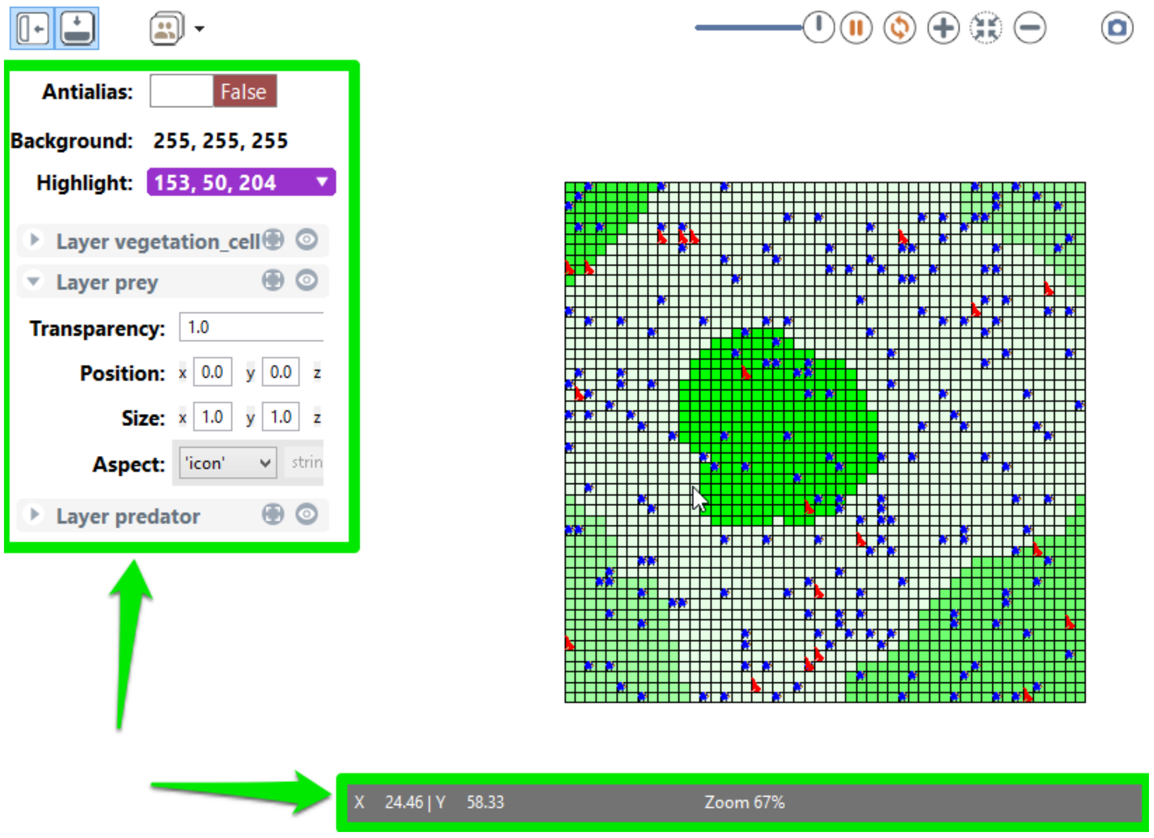
The bottom overlay bar displays information about the way it is displayed: * the position of the mouse in the display, * the zoom ratio, * the scale of the display (depending on the zoom).

OpenGL displays

The OpenGL display has an additional button **3D Options** providing 3D features: * **Use FreeFly camera/Use Arcball camera**: switch between cameras, the default camera is the Arcball one, * **Use mouse to rotate/Use mouse to drag** (only with Arcball camera): use left click for one of the 2 actions, left click + Ctrl for the other of the 2 actions. * **Apply inertia** (only with Arcball camera): in inertia mode, when the modeler stops moving the camera, there is no straight halt but a kind of inertia. * **Rotate scene**: rotate the scene around an axis orthogonal to the scene, * **Split layers/Merge layers**: display each layer at a distinct height, * **Triangulate scene**: display the polygon primitives.

In addition, the bottom overlay bar provides the Camera position in 3D.

FreeFly camera commands


Antialias: False

Background: 255, 255, 255

Highlight: 153, 50, 204

Layer vegetation_cell

Layer prey

Transparency: 1.0

Position: x 0.0 y 0.0 z

Size: x 1.0 y 1.0 z

Aspect: 'icon' strin

Layer predator

X 24.46 | Y 58.33 Zoom 67%

Figure 18.2: images/display-sidebar-overlay.png

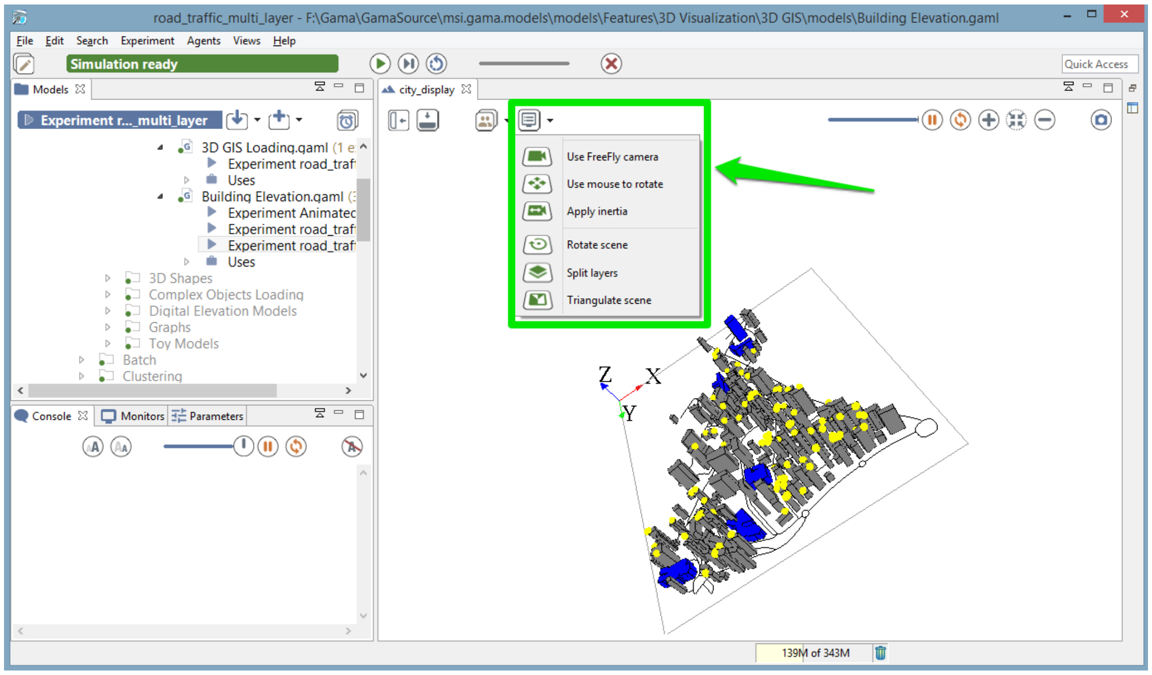


Figure 18.3: images/display-OpenGL.png

Key	**Function**
Double Click	Zoom Fit
+	Zoom In
-	Zoom Out
Up	Move forward
Down	Move backward
Left	Strafe left
Right	Strafe right
SHIFT+Up	Look up
SHIFT+Down	Look down
SHIFT+Left	Look left
SHIFT+Right	Look right
MOUSE	Makes the camera look up, down, left and right
MouseWheel	Zoom-in/out to the current target (center of the screen)

ArcBall camera commands

Key	**Function**
Double Click	Zoom Fit
+	Zoom In
-	Zoom Out
Up	Horizontal movement to the top
Down	Horizontal movement to the bottom
Left	Horizontal movement to the left
Right	Horizontal movement to the right
SHIFT+Up	Rotate the model up (decrease the phi angle of the

Key	**Function**
ALT+LEFT_MOUSE	Enables ROI Agent Selection
SHIFT+LEFT_MOUSE	Enables ROI Zoom
SCROLL	Zoom-in/out to the current target (center of the sphere)
WHEEL CLICK	Reset the pivot to the center of the envelope

Chapter 19

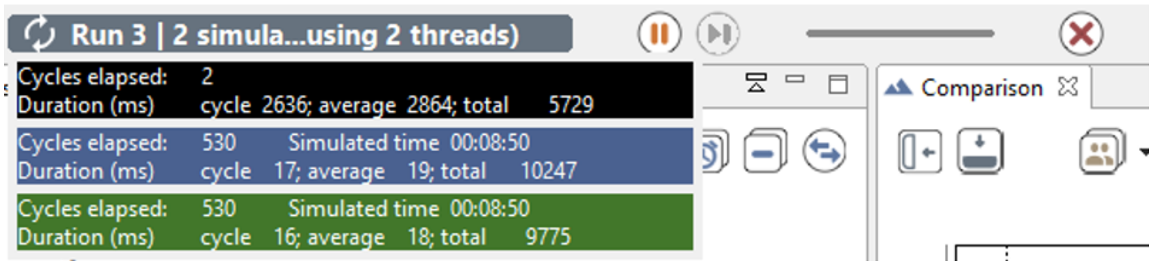
Batch Specific UI

When an experiment of type Batch is run, a dedicated UI is displayed, depending on the parameters to explore and of the exploration methods.

Table of contents

Information bar

In batch mode, the top information bar displays 3 distinct information (instead of only the cycle number in the GUI experiment): * The **run** number: One run corresponds to X executions of simulation with one given parameters values (X is an integer given by the facet `repeat` in the definition of the exploration method); * The **simulation** number: the number of replications done (and the number of replications specified with the `repeat` facet); * The number of **thread**: the number of threads used for the simulation.


Run 3 | 2 simula...using 2 threads)

```
Cycles elapsed:    2
Duration (ms)      cycle 2636; average 2864; total     5729
Cycles elapsed:    530       Simulated time 00:08:50
Duration (ms)      cycle  17; average   19; total    10247
Cycles elapsed:    530       Simulated time 00:08:50
Duration (ms)      cycle  16; average   18; total     9775
```

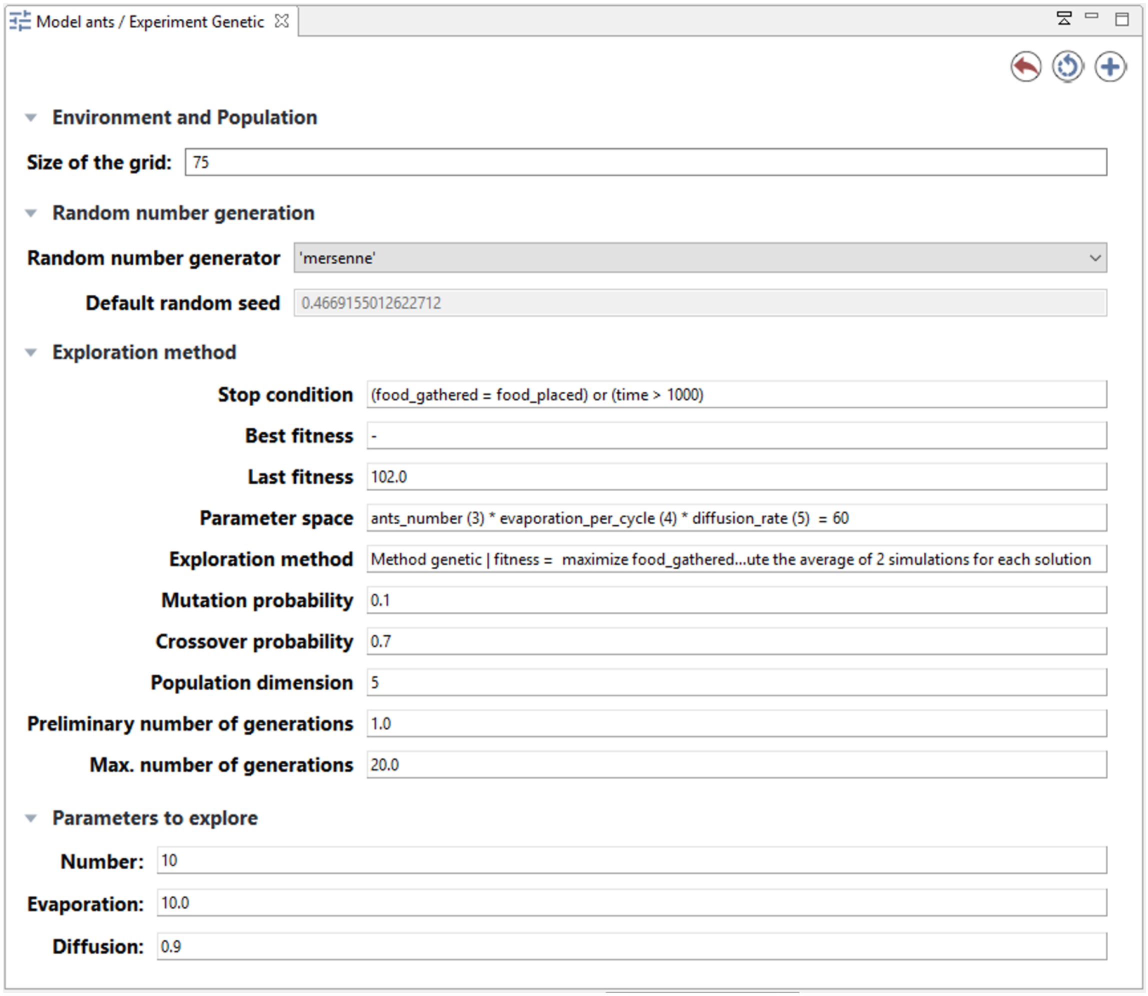
Figure 19.1: images/batch_Information_bar.png

Batch UI

The parameters view is also a bit different in the case of a Batch UI. The following interface is generated given the following model part:

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed) or (time > 400) {
   parameter 'Size of the grid:' var: gridsize init: 75 unit: '
   width and height';
   parameter 'Number:' var: ants_number init: 200 unit: 'ants';
   parameter 'Evaporation:' var: evaporation_rate among: [0.1,
   0.2, 0.5, 0.8, 1.0] unit: 'rate every cycle (1.0 means 100%)';
   parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;

   method exhaustive maximize: food_gathered;
```

The interface summarizes all model parameters and the parameters given to the exploration method: * **Environment and Population**: displays all the model parameters that should not be explored; * **Parameters to explore**: the parameters to explore are the parameters defined in the experiment with a range of values (with `among` facet or `min`, `max` and `step` facets); * **Exploration method**: it summarizes the Exploration method and the stop condition. For exhaustive method it also evaluates the parameter space. For other methods, it also displays the method parameters (e.g. mutation or crossover probability…). Finally the best fitness found and the last fitness found are displayed (with the associated parameter set).


Model ants / Experiment Genetic

Environment and Population

Size of the grid:	75

Random number generation

Random number generator	'mersenne'
Default random seed	0.4669155012622712

Exploration method

Stop condition	(food_gathered = food_placed) or (time > 1000)
Best fitness	-
Last fitness	102.0
Parameter space	ants_number (3) * evaporation_per_cycle (4) * diffusion_rate (5) = 60
Exploration method	Method genetic | fitness = maximize food_gathered...ute the average of 2 simulations for each solution
Mutation probability	0.1
Crossover probability	0.7
Population dimension	5
Preliminary number of generations	1.0
Max. number of generations	20.0

Parameters to explore

Number:	10
Evaporation:	10.0
Diffusion:	0.9

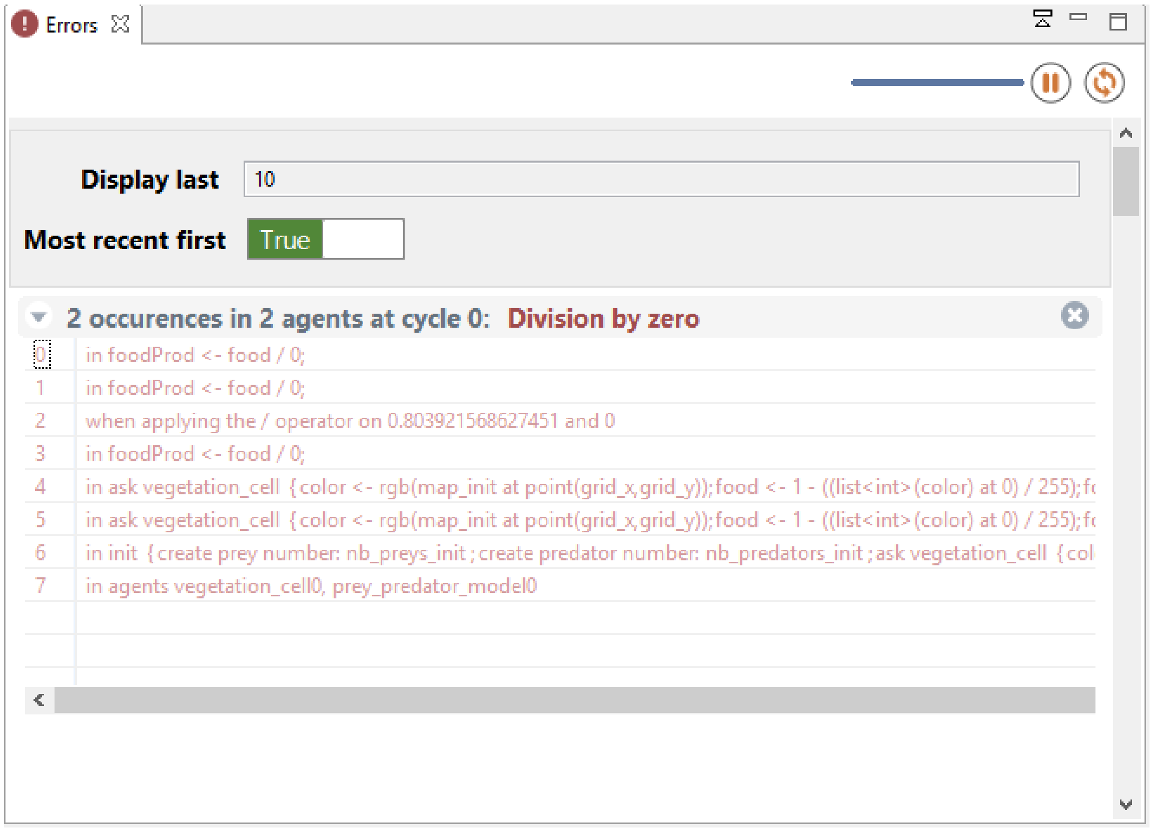
Figure 19.2: images/batch_Parameters_pane.png

Chapter 20

Errors View

Whenever a runtime error, or a warning, is issued by the currently running experiment, a view called "Errors" is opened automatically. This view provides, together with the error/warning itself, some contextual information about who raised the error (i.e. which agent(s)) and where (i.e. in which portion of the model code). As with other "status" in GAMA, errors will appear in red color and warnings in orange.

Since an error appearing in the code is likely to be raised by several agents at once, GAMA groups similar errors together, simply indicating which agent(s) raised them. Note that, unless the error is raised by the experiment agent itself, its message will indicate that at least 2 agents raised it: the original agent and the experiment in which it is plunged.



Display last 10

Most recent first True

```
2 occurences in 2 agents at cycle 0: Division by zero
0 in foodProd <- food / 0;
1 in foodProd <- food / 0;
2 when applying the / operator on 0.803921568627451 and 0
3 in foodProd <- food / 0;
4 in ask vegetation_cell {color <- rgb(map_init at point(grid_x,grid_y));food <- 1 - ((list<int> (color) at 0) / 255);fo
5 in ask vegetation_cell {color <- rgb(map_init at point(grid_x,grid_y));food <- 1 - ((list<int> (color) at 0) / 255);fo
6 in init {create prey number: nb_preys_init ; create predator number: nb_predators_init ; ask vegetation_cell {col
7 in agents vegetation_cell0, prey_predator_model0
```

Figure 20.1: images/errors_view.png

Preferences

Various preferences are accessible in GAMA to allow users and modelers to personalize their working environment. This section review the different preference tabs available in the current version of GAMA, as well as how to access the preferences and settings inherited by GAMA from Eclipse.

Please note that the preferences specific to GAMA will be shared, on a same machine, and for a same user, among all the workspaces managed by GAMA. Changing workspace will not alter them. If you happen to run several instances of GAMA, they will also share these preferences.

Table of contents

Opening Preferences

To open the preferences dialog of GAMA, either click on the small "form" button on the top-left corner of the window or select "Preferences..." from the Gama, "Help" or "Views" menu depending on your OS.

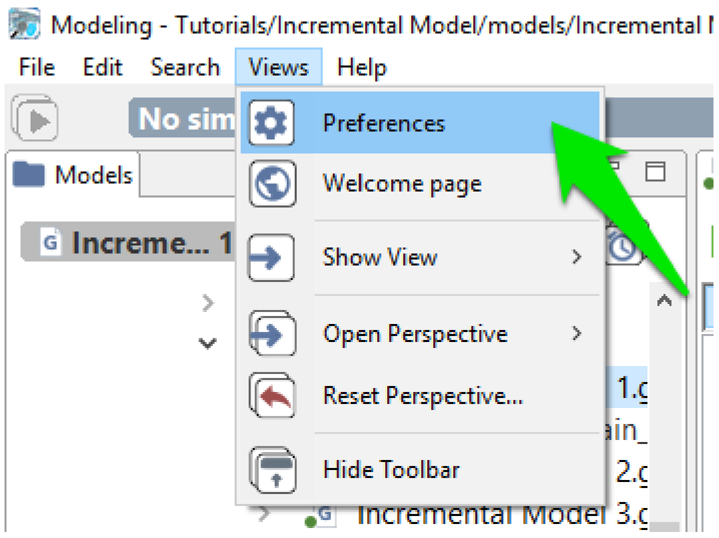
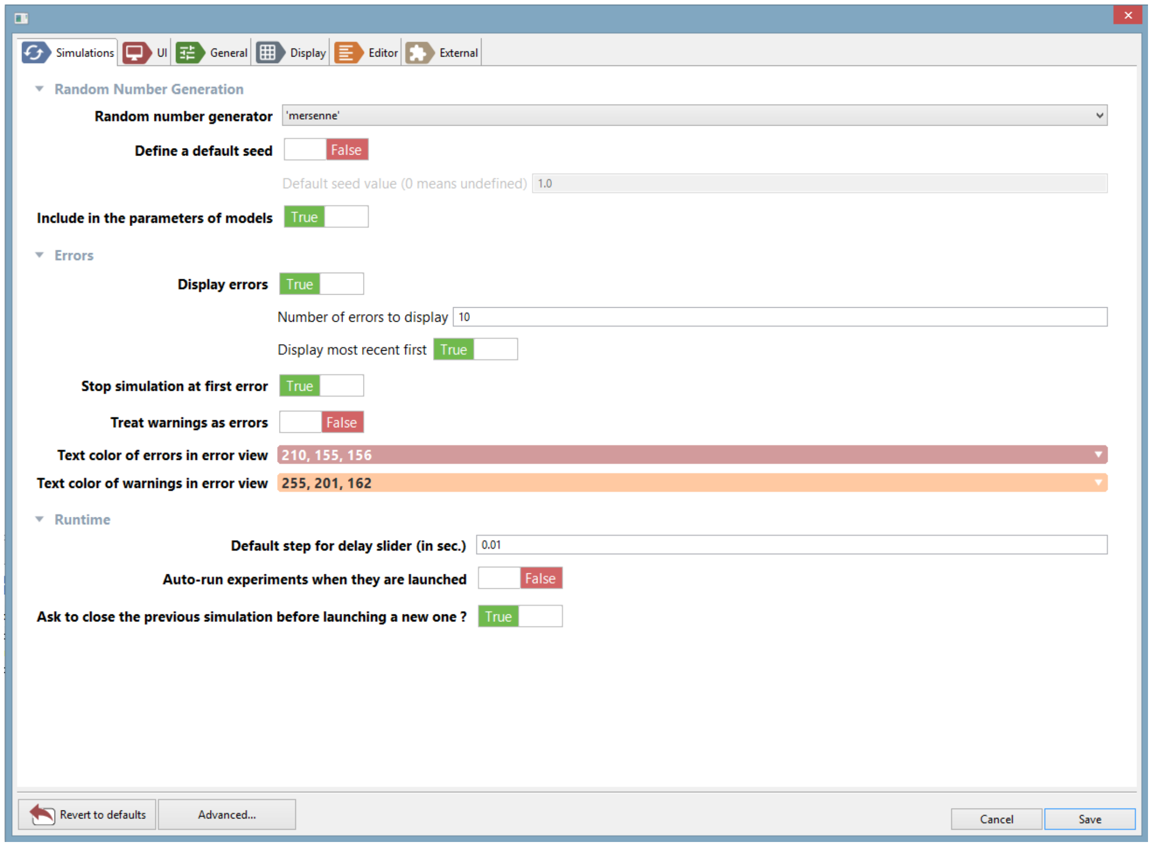


Figure 20.2: images/open_prefs.png

Simulation

- **Random Number Generation**: all the options pertaining to generating random numbers in simulations
  - Random Number Generator: the name of the generator to use by default (if none is specified in the model).
  - Define a default seed: whether or not a default seed should be used if none is specified in the model (otherwise it is chosen randomly by GAMA)
  - Default Seed value: the value of this default seed
  - Include in the parameters of models: whether the choice of generator and seed is included by default in the parameters views of experiments or not.
- **Errors**: how to manage and consider simulation errors
  - Display Errors: whether errors should be displayed or not.
  - Number of errors to display: how many errors should be displayed at once
  - Display most recent first: errors will be sorted in the inverse chronological order if true.
  - Stop simulation at first error: if false, the simulations will display the errors and continue (or try to).
  - Treat warnings as errors: if true, no more distinction is made between warnings (which do not stop the simulation) and errors (which can potentially stop it).



Random Number Generation

Random number generator 'mersenne'

Define a default seed False

Default seed value (0 means undefined) 1.0

Include in the parameters of models True

Errors

Display errors True

Number of errors to display 10

Display most recent first True

Stop simulation at first error True

Treat warnings as errors False

Text color of errors in error view 210, 155, 156

Text color of warnings in error view 255, 201, 162

Runtime

Default step for delay slider (in sec.) 0.01

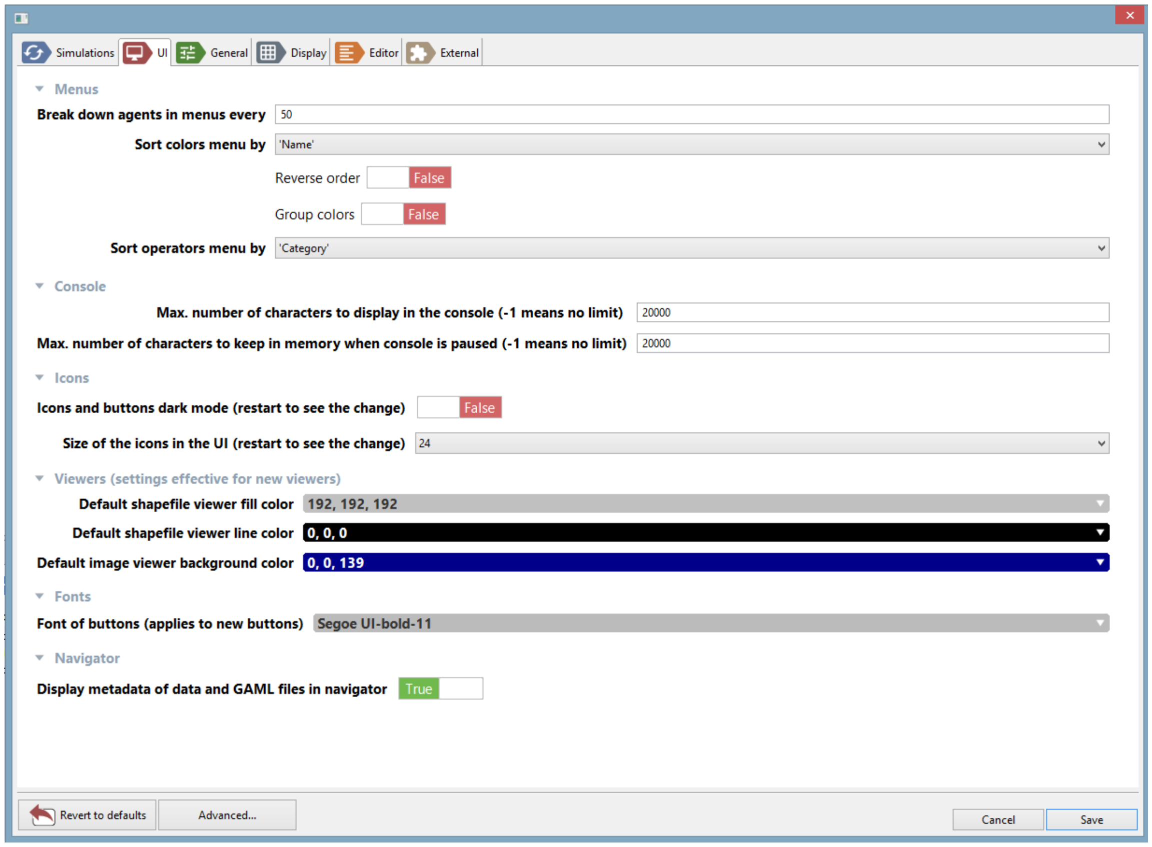
Auto-run experiments when they are launched False

Ask to close the previous simulation before launching a new one ? True

Figure 20.3: images/simulation.png

- **Runtime**: various settings regarding the execution of experiments.
  - Default Step for Delay Slider: the number of seconds that one step of the slider used to impose a delay between two cycles of a simulation lasts.
  - Auto-run experiments when they are launched: see this page.
  - Ask to close the previous simulation before launching a new one: if false, previous simulations (if any) will be closed without warning.

UI



Menus
Break down agents in menus every 50
Sort colors menu by 'Name'
Reverse order False
Group colors False
Sort operators menu by 'Category'
Console
Max. number of characters to display in the console (-1 means no limit) 20000
Max. number of characters to keep in memory when console is paused (-1 means no limit) 20000
Icons
Icons and buttons dark mode (restart to see the change) False
Size of the icons in the UI (restart to see the change) 24
Viewers (settings effective for new viewers)
Default shapefile viewer fill color 192, 192, 192
Default shapefile viewer line color 0, 0, 0
Default image viewer background color 0, 0, 139
Fonts
Font of buttons (applies to new buttons) Segoe UI-bold-11
Navigator
Display metadata of data and GAML files in navigator True

Figure 20.4: images/UI.png

- **Menus**

  - Break down agents in menu every: when inspecting a large number of agents, how many should be displayed before the decision is made to separate the population in sub-menus.
  - Sort colors menu by
  - Sort operators menu by

- **Console**

  - Max. number of characters to display in the console (-1 means no limit)
  - Max. number of characters to keep in memory when console is paused (-1 means no limit)

- **Icons**

  - Icons and buttons dark mode (restart to see the change): Change the highlight for the icons and the button.
  - Size of icons in the UI (restart to see the change): Size of the icons in pixel

- **Viewers**

  - Default shapefile viewer fill color:
  - Default shapefile viewer line color:
  - Default image viewer background color: Background color for the image viewer (when you select an image from the model explorer for example)

General

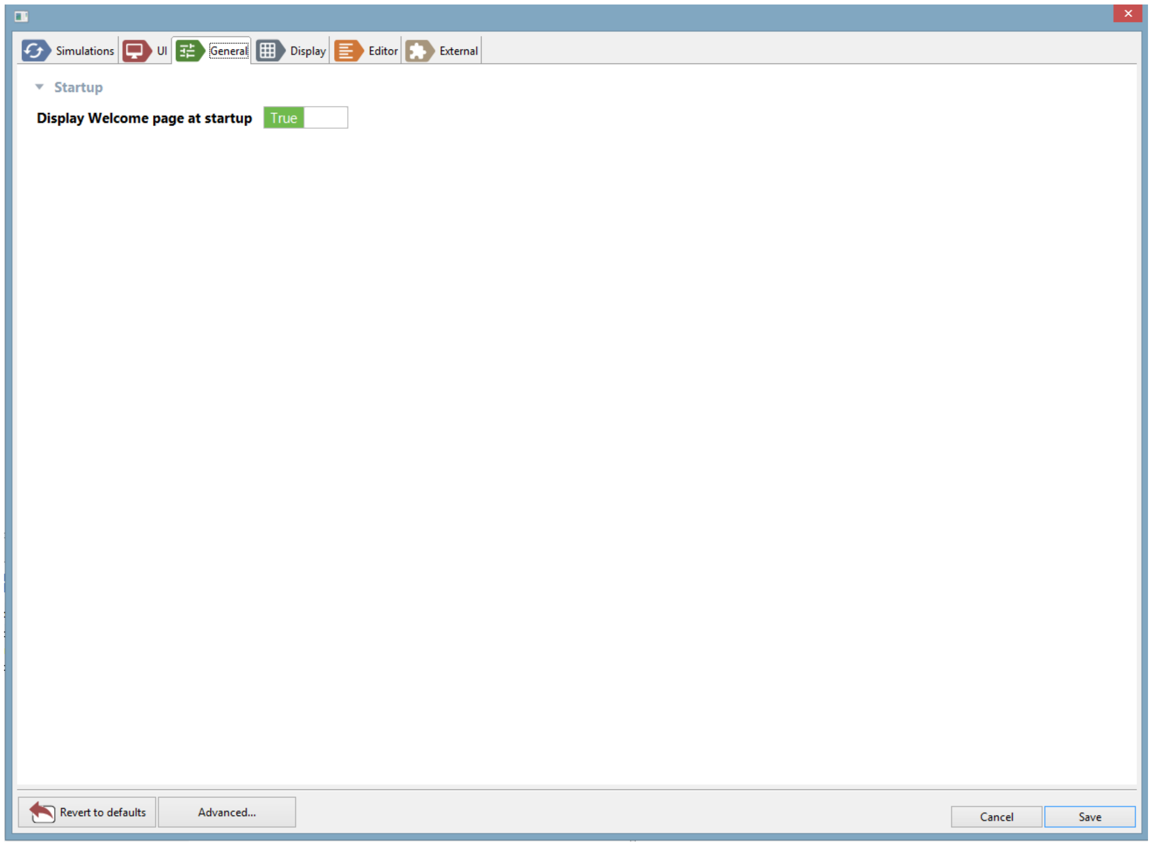
- **Startup**

  - Display welcome page at startup: if true, and if no editors are opened, the welcome page is displayed when opening GAMA.

Display

- **Properties**: various properties of displays

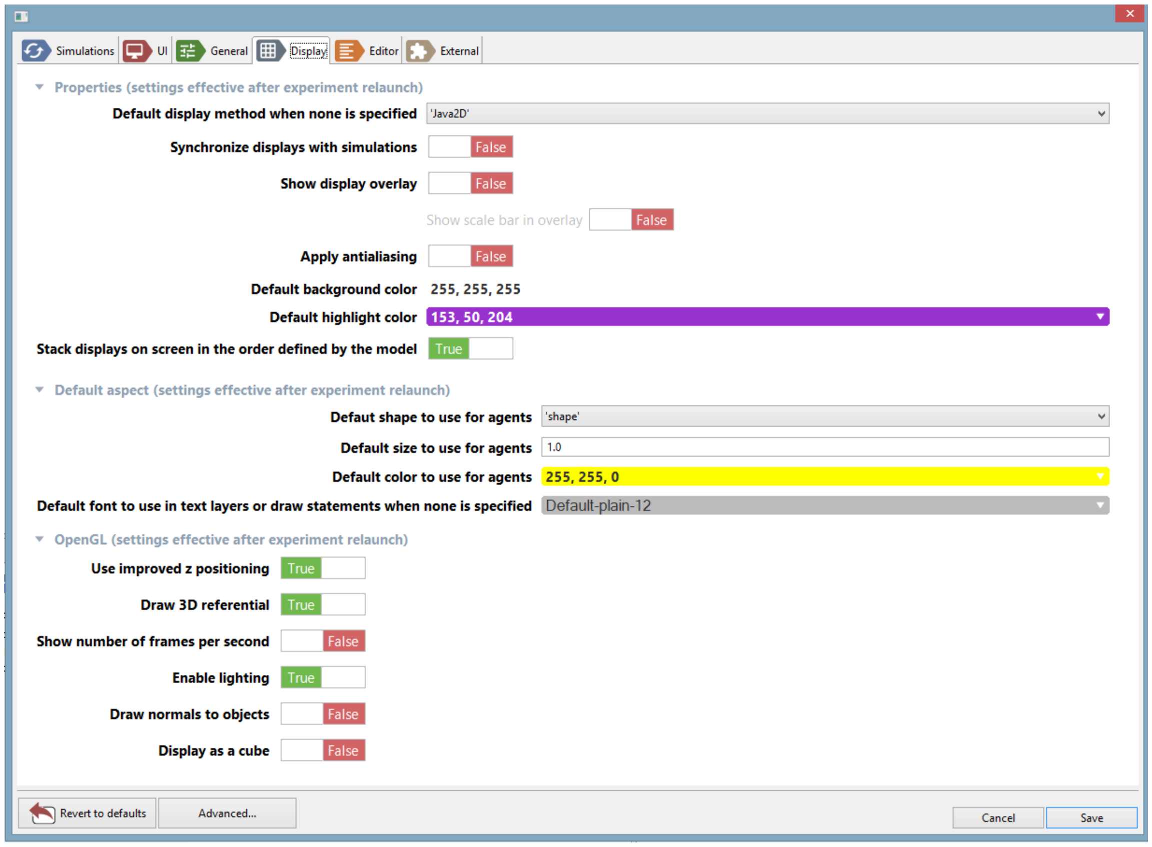
  - Default display method: use either 'Java2D' or 'OpenGL' if nothing is specified in the declaration of a display.



Startup

Display Welcome page at startup True

Figure 20.5: images/general.png



Properties (settings effective after experiment relaunch)

Default display method when none is specified	'Java2D'
Synchronize displays with simulations	False
Show display overlay	False
Show scale bar in overlay	False
Apply antialiasing	False
Default background color	255, 255, 255
Default highlight color	153, 50, 204
Stack displays on screen in the order defined by the model	True

Default aspect (settings effective after experiment relaunch)

Defaut shape to use for agents	'shape'
Default size to use for agents	1.0
Default color to use for agents	255, 255, 0
Default font to use in text layers or draw statements when none is specified	Default-plain-12

OpenGL (settings effective after experiment relaunch)

Use improved z positioning	True
Draw 3D referential	True
Show number of frames per second	False
Enable lighting	True
Draw normals to objects	False
Display as a cube	False

Figure 20.6: images/display.png

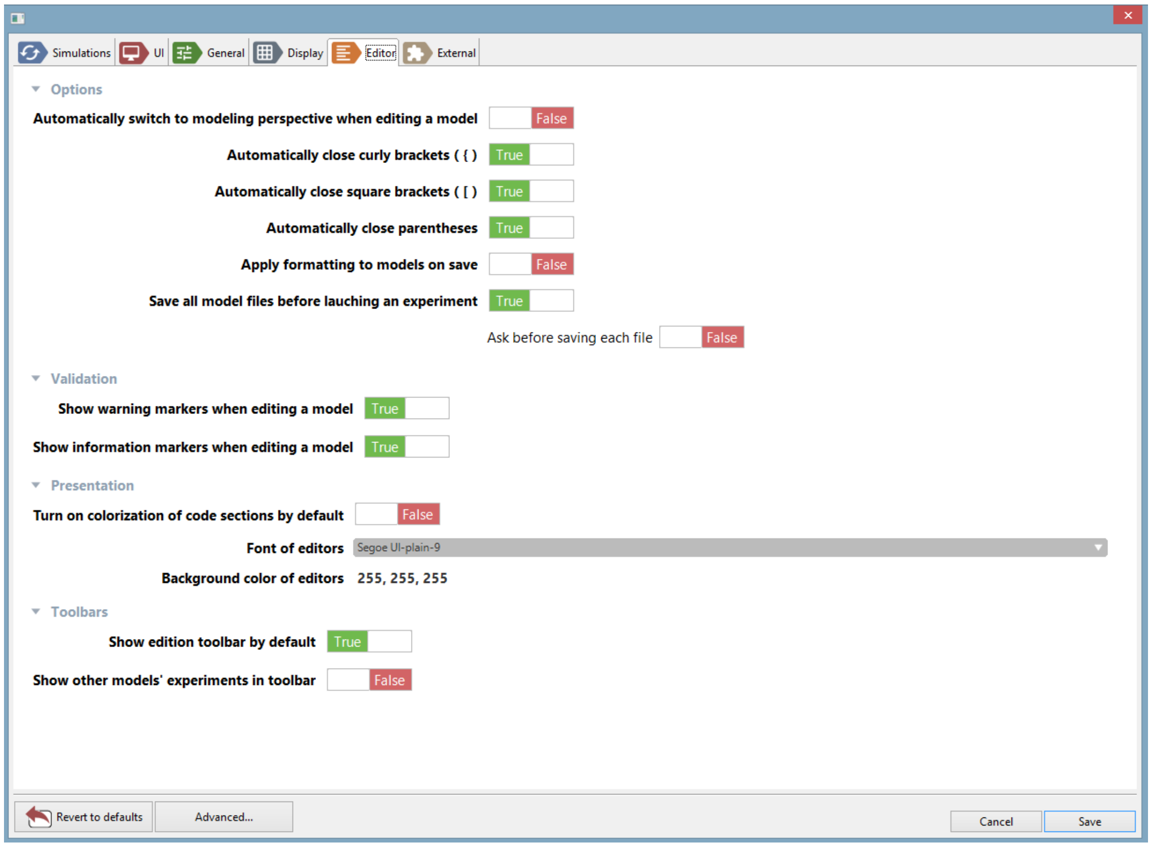
  - Synchronize displays with simulations: if true, simulation cycles will wait for the displays to have finished their rendering before passing to the next cycle (this setting can be changed on an individual basis dynamically here).
  - Show display overlay: if true, the bottom overlay is visible when opening a display.
  - Show scale bar in overlay: if true, the scale bar is displayed in the bottom overlay.
  - Apply antialiasing: if true, displays are drawn using antialiasing, which is slower but renders a better quality of image and text (this setting can be changed on an individual basis dynamically here).
  - Default background color: indicates which color to use when none is specified in the declaration of a display.
  - Default highlight color: indicates which color to use for highlighting agents in the displays.
  - Stack displays on screen…: if true, the display views, in case they are stacked on one another, will put the first display declared in the model on top of the stack.

- **Default Aspect**: which aspect to use when an 'agent' or 'species' layer does not indicate it

  - Default shape: a choice between 'shape' (which represents the actual geometrical shape of the agent) and geometrical operators ('square', etc.).
  - Default size: what size to use. This expression must be a constant.
  - Default color: what color to use.
  - Default font to use in text layers or draw statements when none is specified

- **OpenGL**: various properties specific to OpenGL-based displays

  - Use improved z positioning: if true, two agents positioned at the same z value will be slightly shifted in z in order to draw them more accurately.
  - Draw 3D referential: if true, the shape of the world and the 3 axes are drawn
  - Show number of frames per second
  - Enable lighting: if true, lights can be defined in the display
  - Draw normals to objects: if true, the 'normal' of each object is displayed together with it.
  - Display as a cube: if true, the scene is drawn on all the facets of a cube.



Options

Automatically switch to modeling perspective when editing a model	False
Automatically close curly brackets ({)	True
Automatically close square brackets ([ )	True
Automatically close parentheses	True
Apply formatting to models on save	False
Save all model files before lauching an experiment	True
Ask before saving each file	False

Validation

Show warning markers when editing a model	True
Show information markers when editing a model	True

Presentation

Turn on colorization of code sections by default	False
Font of editors	Segoe UI-plain-9
Background color of editors	255, 255, 255

Toolbars

Show edition toolbar by default	True
Show other models' experiments in toolbar	False

Figure 20.7: images/editor.png

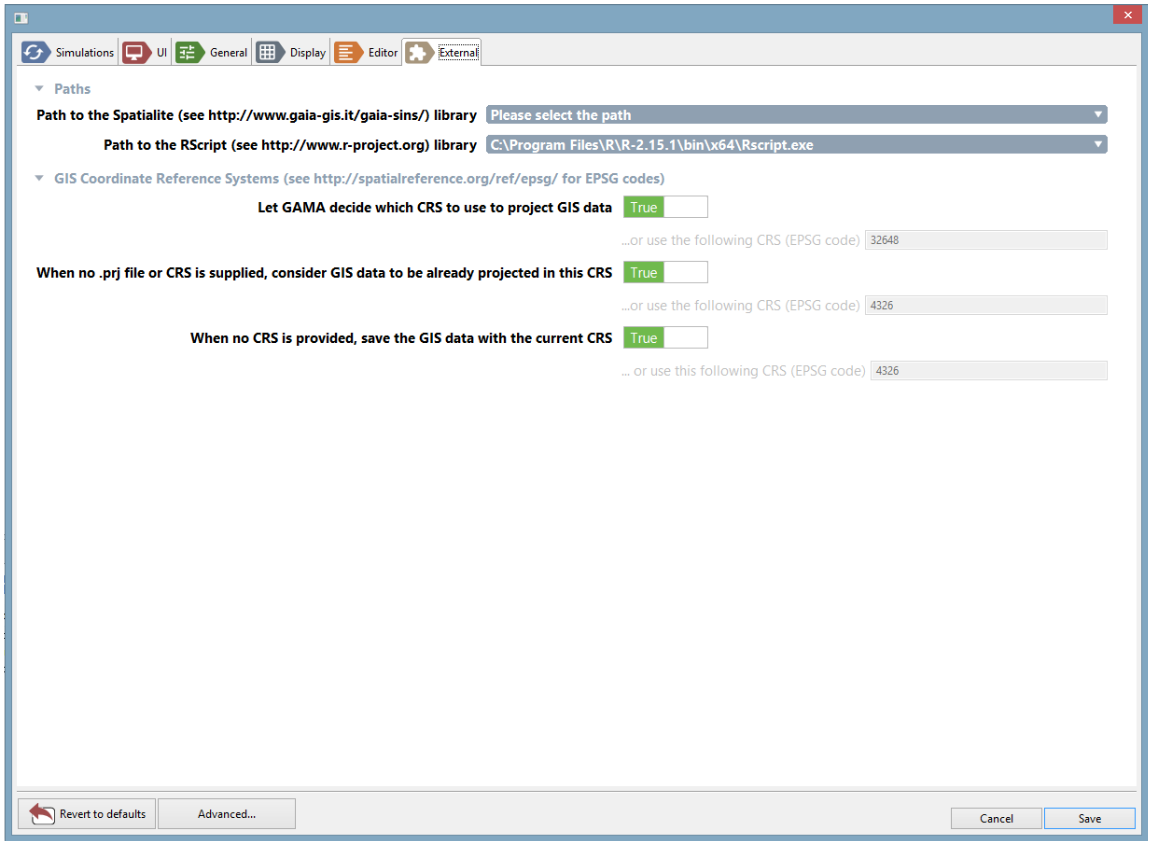
Editor

Most of the settings and preferences regarding editors can be found in the advanced preferences.

- **Options**
  - Automatically switch to Modeling Persepective: if true, if a model is edited in the Simulation Perspective, then the perspective is automatically switched to Modeling (*inactive for the moment*)
  - Automatically close curly brackets ({)
  - Automatically close square brackets (])
  - Automatically close parenthesis
  - Mark occurrences of symbols in models: if true, when a symbol is selected in a model, all its occurrences are also highlighted.
  - Applying formatting to models on save: if true, every time a model file is saved, its code is formatted.
  - Save all model files before launching an experiment
  - Ask before saving each file
- **Validation**
  - Show warning markers when editing a model
  - Show information markers when editing a model
- **Presentation**
  - Turn on colorization of code sections by default
  - Font of editors
  - Background color of editors
- **Toolbars**
  - Show edition toolbar by default
  - Show other models' experiments in toolbar: if true, you are able to launch other models' experiments from a particular model.

External

These preferences pertain to the use of external libraries or data with GAMA.



Paths

Path to the Spatialite (see http://www.gaia-gis.it/gaia-sins/) library: Please select the path

Path to the RScript (see http://www.r-project.org) library: C:\Program Files\R\R-2.15.1\bin\x64\Rscript.exe

GIS Coordinate Reference Systems (see http://spatialreference.org/ref/epsg/ for EPSG codes)

Let GAMA decide which CRS to use to project GIS data: True

...or use the following CRS (EPSG code): 32648

When no .prj file or CRS is supplied, consider GIS data to be already projected in this CRS: True

...or use the following CRS (EPSG code): 4326

When no CRS is provided, save the GIS data with the current CRS: True

... or use this following CRS (EPSG code): 4326

Figure 20.8: images/external.png

- **Paths**
  - Path to Spatialite: the path to the Spatialite library (http://www.gaia-gis.it/gaia-sins/) in the system.
  - Path to RScript: the path to the RScript library (http://www.r-project.org) in the system.
- **GIS Coordinate Reference Systems**: settings about CRS to use when loading or saving GIS files
  - Let GAMA decide which CRS to use to project GIS data: if true, GAMA will decide which CRS, based on input, should be used to project GIS data. Default is false (i.e. only one CRS, entered below, is used to project data in the models)
  - …or use the following CRS (EPSG code): choose a CRS that will be applied to all GIS data when projected in the models. Please refer to http://spatialreference.org/ref/epsg/ for a list of EPSG codes.
  - When no .prj file or CRS is supplied, consider GIS data to be already projected: if true, GIS data that is not accompanied by a CRS information will be considered as projected using the above code.
  - …or use the following CRS (EPSG code): choose a CRS that will represent the default code for loading uninformed GIS data.
  - When no CRS is provided, save the GIS data with the current CRS: if true, saving GIS data will use the projected CRS unless a CRS is provided.
  - …or use the following CRS (EPSG code): otherwise, you might enter a CRS to use to save files.

Advanced Preferences

The set of preferences described above are specific to GAMA. But there are other preferences or settings that are inherited from the Eclipse underpinnings of GAMA, which concern either the "core" of the platform (workspace, editors, updates, etc.) or plugins (like SVN, for instance) that are part of the distribution of GAMA.

These "advanced" preferences are accessible by clicking on the "Advanced…" button in the Preferences view.

Depending on what is installed, the second view that appears will contain a tree of options on the left and preference pages on the right. **Contrary to the first set of preferences, please note that these preferences will be saved in the current workspace**, which

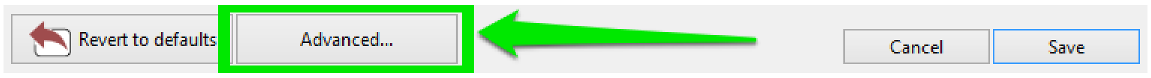
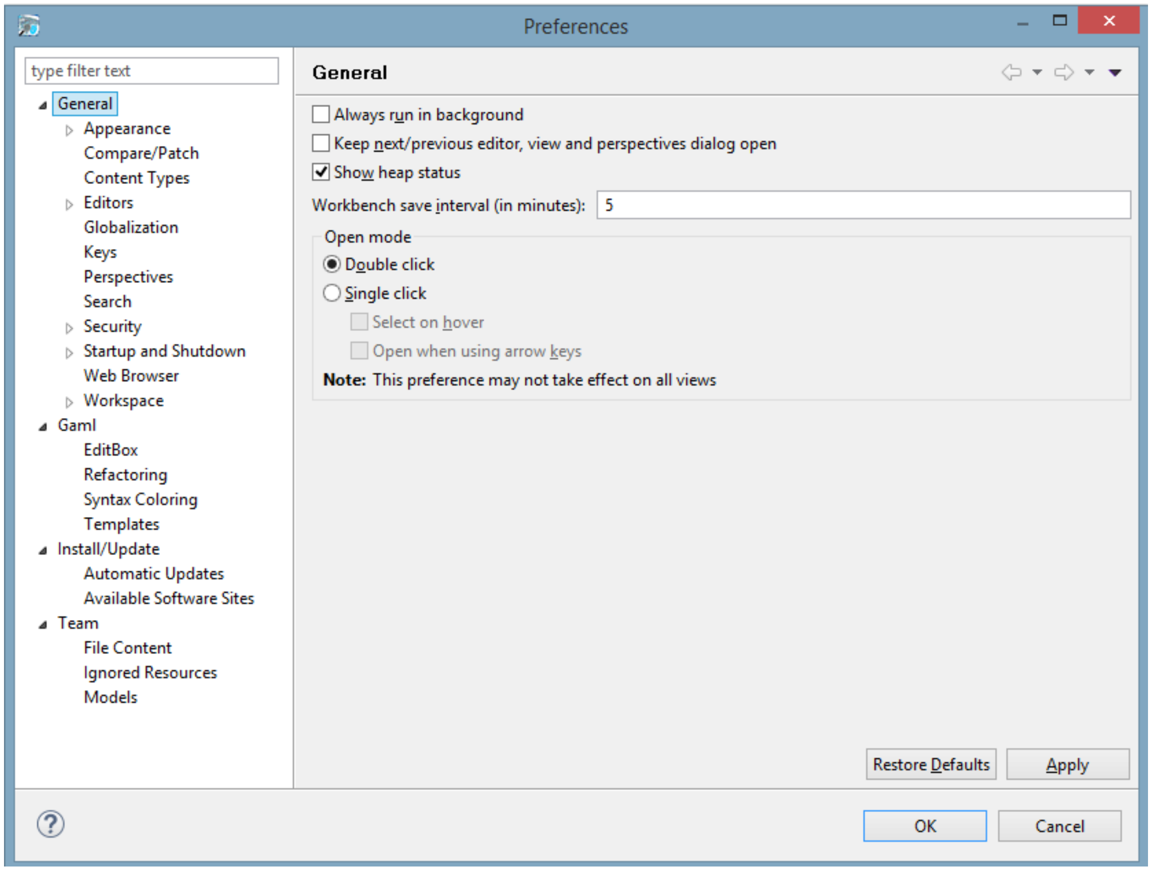


Figure 20.9: images/advanced.png

means that changing workspace will revert them to their default values. It is however possible to import them in the new workspace using of the wizards provided in the standard "Import…" command (see here).



Preferences

type filter text

- General
  - Appearance
  - Compare/Patch
  - Content Types
  - Editors
  - Globalization
  - Keys
  - Perspectives
  - Search
  - Security
  - Startup and Shutdown
  - Web Browser
  - Workspace
- Gaml
  - EditBox
  - Refactoring
  - Syntax Coloring
  - Templates
- Install/Update
  - Automatic Updates
  - Available Software Sites
- Team
  - File Content
  - Ignored Resources
  - Models

General

- [ ] Always run in background
- [ ] Keep next/previous editor, view and perspectives dialog open
- [x] Show heap status

Workbench save interval (in minutes): 5

Open mode
- ● Double click
- ○ Single click
  - [ ] Select on hover
  - [ ] Open when using arrow keys

Note: This preference may not take effect on all views

Figure 20.10: images/advanced_2.png

Part II

GAML (GAMA Modeling Language)

Learn GAML step by step

This large progressive tutorial has been designed to help you to learn **GAML** (**GA**ma **M**odeling **L**anguage). It will cover the main part of the possibilities provided by GAML, and guide you to learn some more.

How to proceed to learn better ?

As you will progress in the tutorial, you will see several links (written in blue) to makes you jump to another part. You can click on them if you want to learn directly about a specific topic, but we do not encourage to do this, because you can get easily lost by reading this tutorial this way. As it is named, we encourage you to follow this tutorial "step by step". For each chapter, some links are available in the "search" tab, if you want to learn more about this subject.

Although, if you really want to learn about a specific topic, our advise is to use the "learning graph" interface, in the website, so that you can choose your area of interest, and a learning path will be automatically design for you to assimilate the specific concept better.

Good luck with your reading, and please do not hesitate to contact us through the mailing list if you have a question/suggestion !

Introduction

GAML is an *agent-oriented* language dedicated to the definition of *agent-based* simulations. It takes its roots in *object-oriented* languages like Java or Smalltalk, but extends the object-oriented programming approach with powerful concepts (like skills, declarative definitions or agent migration) to allow for a better expressivity in models.

It is of course very close to *agent_based* modeling languages like, e.g., NetLogo, but, in addition to enriching the traditional representation of agents with modern computing notions like inheritance, type safety or multi-level agency, and providing the possibility to use different behavioral architectures for programming agents, GAML extends the agent-based paradigm to eliminate the boundaries between the domain of a model (which, in ABM, is represented with agents) and the experimental processes surrounding its simulations (which are usually not represented with agents), including, for example, *visualization* processes. This paper (*Drogoul A., Vanbergue D., Meurisse T., Multi-Agent Based Simulation: Where are the Agents ?, Multi-Agent Based Simulation 3, pp. 1-15, LNCS, Springer-Verlag. 2003*) was in particular foundational in the definition of the concepts on which GAMA (and GAML) are based today.

This orientation has several conceptual consequences among which at least two are of immediate practical interest for modelers: * Since simulations, or experiments, are represented by agents, GAMA is bound to support high-level *model compositionality*, i.e. the definition of models that can use other models as *inner agents*, leveraging multi-modeling or multi-paradigm modeling as particular cases of composition. * The *visualization* of models can be expressed by *models of visualization*, composed of agents entirely dedicated to visually represent other agents, allowing for a clear *separation of concerns* between a simulation and its representation and, hence, the possibility to play with multiple representations of the same model at once.

Table of contents

  - Vocabulary correspondance with the object-oriented paradigm as in Java
  - Vocabulary correspondance with the agent-based paradigm as in NetLogo

Lexical semantics of GAML

The vocabulary of GAML is described in the following sentences, in which the meaning and relationships of the important *words* of the language (in **bold face**) are summarized.

1. The role of GAML is to support modelers in writing **models**, which are specifications of **simulations** that can be executed and controlled during **experiments**, themselves specified by **experiment plans**.
2. The **agent-oriented** modeling paradigm means that everything "active" (entities of a model, systems, processes, activities, like simulations and experiments) can be represented in GAML as an **agent** (which can be thought of as a computational component owning its own data and executing its own behavior, alone or in interaction with other agents).
3. Like in the object-oriented paradigm, where the notion of *class* is used to supply a specification for *objects*, agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they know*), **actions** (*what they can do*), **behaviors** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology** (*how they are connected*) or **schedule** (*in which order and when they should execute*).
4. Any **species** can be nested in another **species** (called its *macro-species*), in which case the **populations** of its instances will imperatively be hosted by an instance of this *macro-species*. A **species** can also inherit its properties from another **species** (called its *parent species*), creating a relationship similar to *specialization* in object-oriented design. In addition to this, **species** can be constructed in a compositional way with the notion of **skills**, bundles of **attributes** and **actions** that can be shared between different species and inherited by their children.
5. Given that all **agents** are specified by a **species**, **simulations** and **experiments** are then instances of two species which are, respectively, called **model** and **experiment plan**. Think of them as "specialized" categories of species.
6. The relationships between **species**, **models** and **experiment plans** are codified in the meta-model of GAML in the form of a framework composed of three abstract species respectively called **agent** (direct or indirect parent of all **species**), **model** (parent of all **species** that define a model) and **experiment** (parent of all **species** that define an experiment plan). In this meta-model, instances of the children of

agent know the instance of the child of **model** in which they are hosted as their **world**, while the instance of **experiment plan** identifies the same agent as one of the **simulations** it is in charge of. The following diagram summarizes this framework:

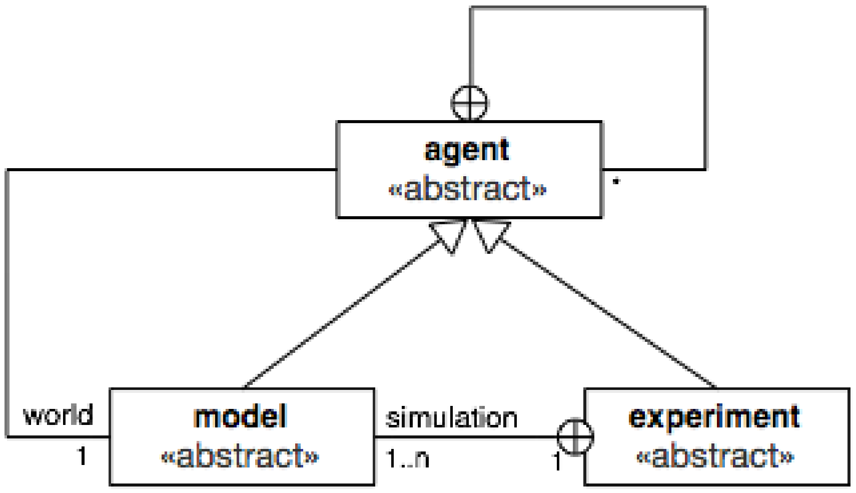


Figure 20.11: framework.png

Putting this all together, writing a model in GAML then consists in defining a species which inherits from **model**, in which other **species**, inheriting (directly or not) from **agent** and representing the entities that populate this model, will be nested, and which is itself nested in one or several **experiment plans** among which a user will be able to choose which **experiment** he/she wants to execute.

At the operational level, i.e. when *running* an experiment in GAMA,

Translation into a concrete syntax

The concepts presented above are expressed in GAML using a syntax which bears resemblances with mainstream programming languages like Java, while reusing some structures from Smalltalk (namely, the syntax of *facets* or the infix notation of *operators*). While this syntax is fully described in the subsequent sections of the documentation, we summarize here the meaning of its most prominent structures and their correspondance (when it exists) with the ones used in Java and NetLogo.

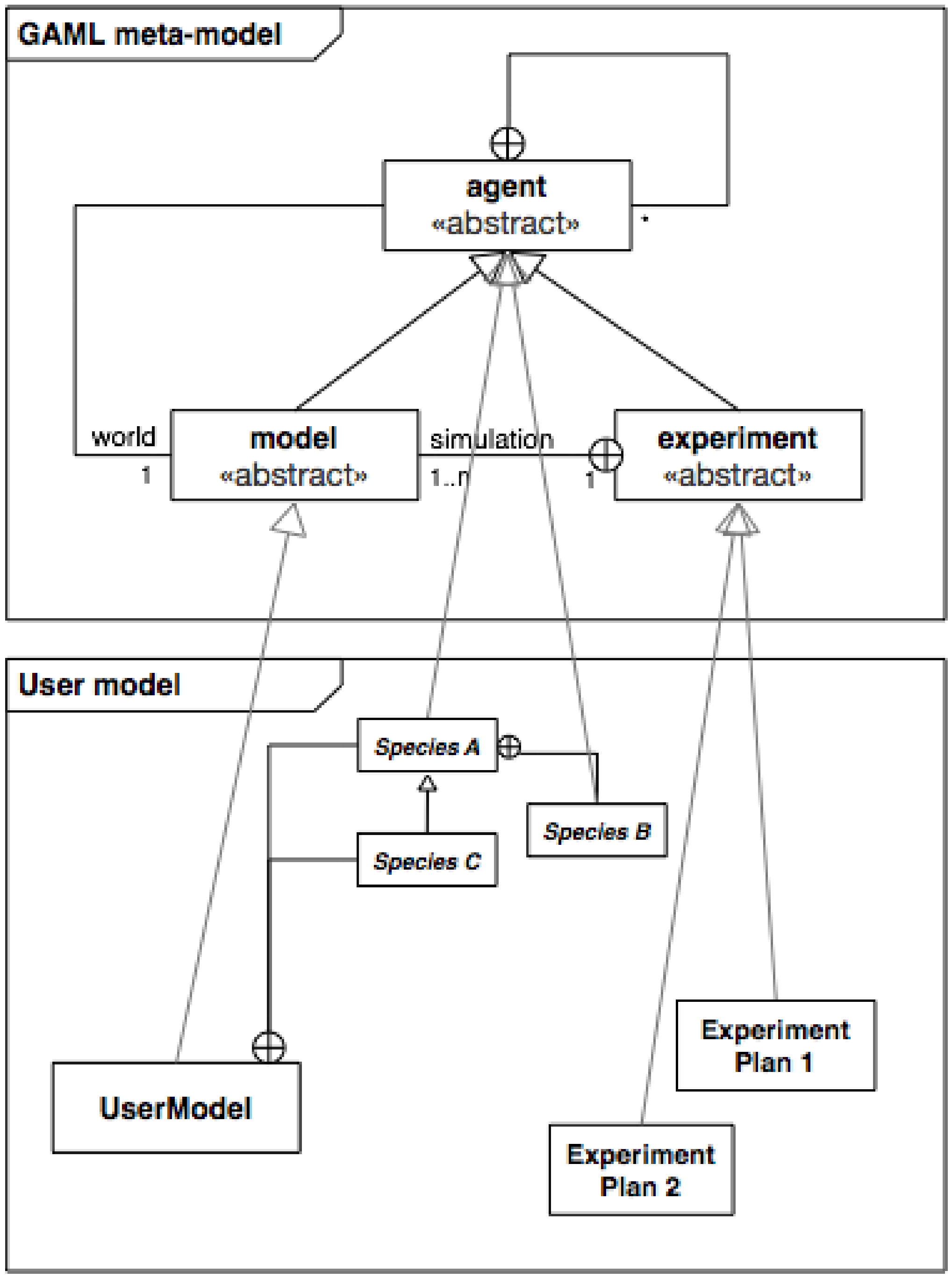


Figure 20.12: user_model.png

1. A **model** is composed of a **header**, in which it can refer to other **models**, and a sequence of **species** and **experiments** declarations, in the form of special **declarative statements** of the language.
2. A **statement** can be either a **declaration** or a **command**. It is always composed of a **keyword** followed by an optional **expression**, followed by a sequence of **facets**, each of them composed of a **keyword** (terminated by a ':') and an **expression**.
3. **facets** allow to pass arguments to **statements**. Their **value** is an **expression** of a given **type**. An **expression** can be a literary constant, the name of an **attribute**, **variable** or **pseudo-variable**, the name of a **unit** or **constant** of the language, or the application of an **operator**.
4. A **type** can be a **primitive type**, a **species type** or a **parametric type** (i.e. a composition of **types**).
5. Some **statements** can include sub-statements in a **block** (sequence of **statements** enclosed in curly brackets).
6. **declarative statements** support the definition of special constructs of the language: for instance, **species** (including **global** and **experiment** species), **attributes**, **actions**, **behaviors**, **aspects**, **variables**, **parameters** and **outputs** of **experiments**.
7. **imperative statements** that execute something or control the flow of execution of **actions**, **behaviors** and **aspects** are called **commands**.
8. A **species** declaration (**global**, **species** or **grid** keywords) can only include 6 types of declarative statements : **attributes**, **actions**, **behaviors**, **aspects**, **equations** and (nested) **species**. In addition, **experiment** species allow to declare **parameters**, **outputs** and batch **methods**.

Vocabulary correspondance with the object-oriented paradigm as in Java

GAML	Java
species	class
micro-species	nested class
parent species	superclass
child species	subclass
model	program
experiment	(main) class
agent	object

GAML	Java
attribute	member
action	method
behavior	collection of methods
aspect	collection of methods, mixed with the behavior
skill	interface (on steroids)
statement	statement
type	type
parametric type	generics

Vocabulary correspondance with the agent-based paradigm as in NetLogo

GAML	NetLogo
species	breed
micro-species	-
parent species	-
child species	- (only from 'turtle')
model	model
experiment	observer
agent	turtle/observer
attribute	'breed'-own
action	global function applied only to one breed
behavior	collection of global functions applied to one breed
aspect	only one, mixed with the behavior
skill	-
statement	primitive
type	type
parametric type	-

Start with GAML

In this part, we will present you some basic concepts of GAML that will help you a lot for the next pages.

You will first learn how to **organize a standard model**, then you will learn about some **basis about GAML**, such as how to declare a variable, how to use the basic operators, how to write a conditional structure or a loop, how to manipulate containers and how to generate random values.

Chapter 21

Organization of a model

As already extensively detailed in the introduction page, defining a model in GAML amounts to defining a *model species*, which later allows to instantiate a *model agent* (aka a *simulation*), which may or may not contain micro-species, and which can be flanked by *experiment plans* in order to be simulated.

This conceptual structure is respected in the definition of model files, which follows a similar pattern:

1. Definition of the *global species*, preceded by a *header*, in order to represent the *model species*
2. Definition of the different micro-species (either nested inside the *global species* or at the same level)
3. Definition of the different *experiment plans* that target this model

Table of contents

Model Header (*model species*)

The header of a model file begins with the declaration of the name of the model. Contrarily to other statements, this declaration **does not** end with a semi-colon.

```
model name_of_the_model
```

The name of the model is not necessarily the same as the name of the file. It must conform to the general rule for naming species, i.e. be a valid identifier (beginning with a letter, containing only letters, digits and dashes). This name will be used for building the name of the model species, from which *simulations* will be instantiated. For instance, the following declaration:

```
model dummy
```

will internally create a species called `dummy_model`, child of the abstract species `model`, from which simulations (called `dummy_model0`, `dummy_model1`, etc.) will be instantiated.

This declaration is followed by optional import statements that indicate which other models this model is importing. Import statements **do not** end with a semi-colon.

Importing a model can take two forms. The first one, called *inheritance import*, is declared as follows:

```
import "relative_path_to_a_model_file"
import "relative_path_to_another_model_file"
```

The second one, called *usage import*, is declared as follows:

```
import "relative_path_to_a_model_file" as model_identifier
```

When importing models using the first form, all the declarations of the model(s) imported will be merged with those of the current model (in the order with which the import statements are declared, i.e. the latest definitions of global attributes or behaviors superseding the previous ones). The second form is reserved for using models as *micro-models* of the current model. This possibility is still experimental in the current version of GAMA.

The last part of the *header* is the definition of the `global` species, which is the actual definition of the *model species* itself.

```
global {
    // Definition of [global attributes](GlobalSpecies#
   declaration), [actions and behaviors](
   DefiningActionsAndBehaviors)
}
```

Note that neither the imports nor the definition of `global` are mandatory. Only the `model` statement is.

Species declarations

The header is followed by the declaration of the different species of agents that populate the model.

The special species `global` is the world species. You will declare here all the global attributes/actions/behaviors. The global species does not have name, and is unique in your model.

```
global {
    // definition of global attributes, actions, behaviors
}
```

Regular species can be declared with the keyword `species`. You can declare several regular species, and they all have to be named.

```
species nameOfSpecies {
    // definition of your [species attributes](RegularSpecies#
   declaration), [actions and behaviors](
   DefiningActionsAndBehaviors)
}
```

Note that the possibility to define the species *after* the `global` definition is actually a convenience: these species are micro-species of the model species and, hence, could be perfectly defined as nested species of `global`. For instance:

```
global {
    // definition of global attributes, actions, behaviors
}

species A …{}
species B …{}
```

is completely equivalent to:

```
global {
    // definition of [global attributes](GlobalSpecies#
   declaration), actions, behaviors

    species A …{}

    species B …{}
}
```

Experiment declarations

Experiments are usually declared at the end of the file. They start with the keyword `experiment`. They contains the simulation parameters, and the definition of the output (such as displays, monitors or inspectors). You can declare as much experiments as you want.

```
experiment first_experiment {
    // definition of parameters (intputs)

    // definition of output
    output {...}
}

experiment second_experiment {
    // definition of parameters (inputs)

    // definition of output
}
```

Note that you have two types of experiments: A GUI experiment allows to display a graphical interface with input parameters and outputs. It is declared with the following structure :

```
experiment gui_experiment type:gui {
    [...]
}
```

A Batch experiment allows to execute numerous successive simulation runs (often used for model exploration). It is declared with the following structure :

```
experiment batch_experiment type:batch {
    [...]
}
```

Basic skeleton of a model

Here is the basic skeleton of a model :

```
model name_of_the_model

global {
    // definition of [global attributes](GlobalSpecies#
   declaration), actions, behaviours
}

species my_specie {
    // definition of attributes, actions, behaviours
}

experiment my_experiment /* + specify the type : "type:gui" or "
   type:batch" */
{
    // here the definition of your experiment, with...
    // ... your inputs
    output {
        // ... and your outputs
    }
}
```

Don't forget this structure ! This will be the basis for all the models you will create from now. //: # (endConcept|model_structure)

Chapter 22

Basic programming concepts in GAML

In this part, we will focus on the very basic structures in GAML, such as how to declare a variable, how to use loops, or how to manipulate lists. We will overfly quickly all those basic programming concepts, admitting that you already have some basics in coding.

Index

Variables

Variables are declared very easily in GAML, starting with the keyword for the type, following by the name you want for your variable. NB: The declaration has to be inside the `global` scope, or inside the `species` scope.

```
typeName myVariableName;
```

Basic types

All the "basic" types are present in GAML:`int`, `float`, `string`, `bool`. The operator for the affectation in GAML is <- (the operator = is used to test the equality).

```
int integerVariable <- 3;
float floatVariable <- 2.5;
string stringVariable <- "test"; // you can also write simple ' :
    <- 'test'
bool booleanVariable <- true; // or false
```

To follow the behavior of variable, we can write their value in the console. Let's go back to our basic skeleton of a model, and let's create a reflex in the global scope (to be short, a reflex is a function that is executed in each step. We will come back to this concept later). The `write` function works very easily, simply writing down the keyword `write` and the name of the variable we want to be displayed.

```
model firstModel

global {
    int integerVariable <- 3;
    float floatVariable <- 2.5;
    string stringVariable <- "test"; // you can also write simple
    ' : <- 'test'
    bool booleanVariable <- true; // or false
    reflex writeDebug {
        write integerVariable;
        write floatVariable;
        write stringVariable;
        write booleanVariable;
    }
}

experiment myExperiment
{
}
```

The function `write` is overloaded for each type of variable (even for the more complex type, such as containers).

Note that before being initialized, a variable has the value `nil`.

```
reflex update {
    string my_string;
    write my_string; // this will write "nil".
    int my_int;
    write my_int; // this will write "0", which is the default
   value for int.
}
```

`nil` is also a literal you can use to initialize your variable (you can learn more about the concept of literal in this page).

```
reflex update {
    string my_string <- "a string";
    my_string <- nil;
    write my_string; // this will write "nil".
    int my_int <- 6;
    my_int <- nil;
    write my_int; // this will write "0", which is the default
   value for int.
}
```

The point type

Another variable type you should know is the point variable. This type of variable is used to describe coordinates. It is in fact a complex variable, composed of two float variables (or three if you are working in 3D). To declare it, you have to use the curly bracket {:

```
point p <- {0.2,2.4};
```

The first field is related to the x value, and the second, to the y value. You can easily get this value as following:

```
point p <- {0.2,2.4};
write p.x; // the output will be 0.2
write p.y; // the output will be 2.4
```

You can't modify directly the value. But if you want, you can do a simple operation to get what you want:

```
point p <- {0.2,2.4};
p <- p + {0.0,1.0};
write p.y; // the output will be 3.4
```

A world about dimensions

When manipulating float values, you can specify the dimension of your value. Dimensions are preceded by # or ° (exactly the same).

```
float a <- 5°m;
float b <- 4#cm;
float c <- a + b; // c is equal to 5.0399999 (it's not equal to
   5.04 because it is a float value, not as precise as int)
```

Declare variables using facet

Facets are used to describe the behavior of a variable during its declaration, by adding the keyword `facet` just after the variable name, followed by the value you want for the facet (or also just after the initial value).

```
type variableName <- initialValue facet1:valueForFacet1 facet2:
   valueForFacet2;
// or:
type variableName facet1:valueForFacet1 facet2:valueForFacet2;
variableName <- initialValue;
```

You can use the facet `update` if you want to change the value of your variable. For example, to increment your integer variable each step, you can do as follow:

```
int integerVariable <- 3 min:0 max:10 update:integerVariable+1;
// nb: the operator "++" doesn't exist in gaml.
```

You can use the facet `min` and `max` to constraint the value in a specific range of values:

```
int integerVariable <- 3 min:0 max:10 update:integerVariable+1;
// the result will be 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 10 - 10 -
   ...
```

The facet `among` can also be useful (that can be seen as an enum):

```
string fruits <- "banana" among:["pear","apple","banana"];
```

Operators in GAMA

In GAML language, you can use a lot of different operators. They are all listed in this page, but here are the most useful ones:

- Mathematical operators

The basic arithmetical operators, such as +(add), -(substract), ∗(multiply), /(divide), ˆ (power) are used this way:

FirstOperand Operator SecondOperand –> ex: 5 * 3; // return 15

Some other operators, such as `cos`(cosinus), `sin`(sinus), `tan`(tangent), `sqrt`(square root), `round`(rounding) etc... are used this way:

```
Operator(Operand) --> ex: sqrt(49); // return 7
```

Logical operators

Logical operators such as `and`(and), `or`(inclusive or) are used the same way as basic arithmetical operators. The operator `!`(negation) has to be placed just before the operand. They return a boolean result.

```
FirstOperand Operator SecondOperand --> ex: true or false; //
   return true
NegationOperator Operand --> ex: !(true or false); // return
   false
```

Comparison operators

The comparison operators `!=`(different than), <(smaller than), <=(smaller of equal), = (equal), >(bigger than), >=(bigger or equal) are used the same way as basic arithmetical operators:

```
FirstOperand Operator SecondOperand --> ex: 5 < 3; // return
   false
```

Type casting operators

You can cast an operand to a special type using casting operator:

```
Operator(Operand); --> ex: int(2.1); // return 2
```

Other operators

A lot of other operators exist in GAML. The standard way to use those operators is as followed:

```
Operator(FirstOperand,SecondOperand,...) --> ex: rnd(1,8);
```

Some others are used in a more intuitive way:

```
FirstOperand Operator SecondOperand --> ex: 2[6,4,5] contains(5);
```

Conditional structures

You can write if/else if/else in GAML:

```
if (integerVariable<0) {
    write "my value is negative !! The exact value is " +
   integerVariable;
}
else if (integerVariable>0) {
    write "my value is positive !! The exact value is " +
   integerVariable;
}
else if (integerVariable=0) {
    write "my value is equal to 0 !!";
}
else {
    write "hey... This is not possible, right ?";
}
```

GAML also accepts ternary operator:

```
stringVariable <- (booleanVariable) ? "booleanVariable = true" :
   "booleanVariable = false";
```

Loop

Loops in GAML are designed by the keyword `loop`. As for variables, a loop have multiple facet to determine its behavior:

- The facet `times`, to repeat a fixed number of times a set of statements:

```
loop times: 2 {
write "helloWorld";
}
// the output will be helloWorld - helloWorld
```

- The facet `while`, to repeat a set of statements while a condition is true:

```
loop while: (true) {
}
// infinity loop
```

- The facet `from` / `to`, to repeat a set of statements while an index iterates over a range of values with a fixed step of 1:

```
loop i from:0 to: 5 { // or loop name:i from:0 to:5 -> the name
    is also a facet
     write i;
}
// the output will be 0 - 1 - 2 - 3 - 4 - 5
```

- The facet `from` / `to` combine with the facet `step` to choose the step:

```
loop i from:0 to: 5 step: 2 {
     write i;
}
// the output will be 0 - 2 - 4
```

- The facet over to browse containers, as we will see in the next part.

Nb: you can interrupt a loop at any time by using the `break` statement.

Manipulate containers

We saw in the previous parts "simple" types of variable. You also have a multiple containers types, such as list, matrix, map, pair… In this section, we will only focus on the container `list` (you can learn the other by reading the section about datatypes).

How to declare a list?

To declare a list, you can either or not specify the type of the data of its elements:

```
list<int> listOfInt <- [5,4,9,8];
list listWithoutType <- [2,4.6,"oij",["hoh",0.0]];
```

How to know the number of elements of a list?

To know the number of element of a list, you can use the operator `length` that returns the number of elements (note that this operator also works with strings).

```
int numberOfElements <- length([12,13]); // will return 2
int numberOfElements <- length([]); // will return 0
int numberOfElements <- length("stuff"); // will return 5
```

There is an other operator, `empty`, that returns you a boolean telling you if the list is empty or not.

```
bool isEmpty <- empty([12,13]); // will return false
bool isEmpty <- empty([]); // will return true
bool isEmpty <- empty("stuff"); // will return false
```

How to get an element from a list?

To get an element from a list by its index, you have to use the operator `at` (nb: it is indeed an operator, and not a facet, so no ":" after the keyword).

```
int theFirstElementOfTheList <- [5,4,9,8] at 0; // this will
   return 5
int theThirdElementOfTheList <- [5,4,9,8] at 2; // this will
   return 9
```

How to know the index of an element of a list?

You can know the index of the first occurrence of a value in a list using the operator `index_of`. You can know the index of the last occurrence of a value in a list using the operator `last_index_of`.

```
int result <- [4,2,3,4,5,4] last_index_of 4;  // result equals 5
int result <- [4,2,3,4,5,4] index_of 4;  // result equals 0
```

How to know if an element exists in a list?

You can use the operator `contains` (return a boolean):

```
bool result <- [{1,2}, {3,4}, {5,6}] contains {3,4};  // result
   equals true
```

How to insert/remove an element to/from a list?

For those operation, no operator are available, but you can use a statement instead. The statements `add` and `put` are used to insert/modify an element, while the statement `remove` is used to remove an element. Here are some example of how to use those 3 statements with the most common facets:

```
list<int> list_int <- [1,5,7,6,7];
remove from:list_int index:1; // remove the 2nd element of the
   list
write list_int; // the output is : [1,7,6,7]
```

```
remove item:7 from:list_int; // remove the 1st occurrence of 7
write list_int; // the output is : [1,6,7]
add item:9 to: list_int at: 2; // add 9 in the 3rd position
write list_int; // the output is : [1,6,9,7]
add 0 to: list_int; // add 0 in the last position
write list_int; // the output is : [1,6,9,7,0]
put 3 in: list_int at: 0; // put 3 in the 1st position
write list_int; // the output is : [3,6,9,7,0]
put 2 in: list_int key: 2; // put 2 in the 3rd position
write list_int; // the output is : [3,6,2,7,0]
```

How to add 2 lists?

You can add 2 lists by creating a third one and browsing the 2 first one, but you can do it much easily by using the operator + :

```
list<int> list_int1 <- [1,5,7,6,7];
list<int> list_int2 <- [6,9];
list<int> list_int_result <- list_int1 + list_int2;
```

How to browse a list?

You can use the facet `over` of a loop:

```
list<int> exampleOfList <- [4,2,3,4,5,4];
loop i over:exampleOfList {
    write i;
}
// the output will be 4 - 2 - 3 - 4 - 5 - 4
```

How to filter a list?

If you want to get all the elements of a list that fulfill a particular condition, you need the operator where. In the condition, you can design all the element of a particular list by using the pseudo variable `each` as followed:

```
list<int> exampleOfList <- [4,2,3,4,5,4] where (each <= 3);
// the list is now [2,3]
```

Other useful operators for the manipulation of lists:

Here are some other operators which can be useful to manipulate lists: `sort`, `sort_by`, `shuffle`, `reverse`, `collect`, `accumulate`, `among`. Please read the GAML Reference if you want to know more about those operators.

Random values

When you will implement your model, you will have to manipulate some random values quite often.

To get a random value in a range of value, use the operator `rnd`. You can use this operator in many ways:

```
int var0 <- rnd (2);    // var0 equals 0, 1 or 2
float var1 <- rnd (1000) / 1000;    // var1 equals a float
   between 0 and 1 with a precision of 0.001
point var2 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0}, 1);    // var2
   equals a point with x = 2.0, y equal to 2.0, 3.0 or 4.0 and z
   between 0.0 and 10.0 every 1.0
float var3 <- rnd (2.0, 4.0, 0.5);  // var3 equals a float number
    between 2.0 and 4.0 every 0.5
float var4 <- rnd(3.4);     // var4 equals a random float between
    0.0 and 3.4
int var5 <- rnd (2, 12, 4);     // var5 equals 2, 6 or 10
point var6 <- rnd ({2.5,3, 0.0});   // var6 equals {x,y} with x
   in [0.0,2.0], y in [0.0,3.0], z = 0.0
int var7 <- rnd (2, 4);     // var7 equals 2, 3 or 4
point var8 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0});   // var8
   equals a point with x = 2.0, y between 2.0 and 4.0 and z
   between 0.0 and 10.0
float var9 <- rnd (2.0, 4.0);   // var9 equals a float number
   between 2.0 and 4.0
```

Use the operator `flip` if you want to pick a boolean value with a certain probability:

```
bool result <- flip(0.2); // result will have 20% of chance to be
    true
```

You can use randomness in list, by using the operator `shuffle`, or also by using the operator `among` to pick randomly one (or several) element of your list:

```
list TwoRandomValuesFromTheList <- 2 among [5,4,9,8];
// the list will be for example [5,9].
```

You can use probabilistic laws, using operators such as `gauss`, `poisson`, `binomial`, or `truncated_gauss` (we invite you to read the documentation for those operators). //: # (endConcept|programming_basis)

Manipulate basic species

In this chapter, we will learn how to manipulate some basic species. As you already know, a species can be seen as the definition of a type of **agent** (we call agent the instance of a species). In OOP (Object-Oriented Programming), a **species** can be seen as the class. Each species is then defined by some **attributes** ("member" in OOP), **actions** ("method" in OOP) and **behavior** ("method" in OOP).

In this section, we will first learn how to declare the **world agent**, using the **global species**. We will then learn how to declare **regular species** which will populate our world. The following lesson will be dedicated to learn how to **define actions and behaviors** for all those species. We will then learn how **agents can interact between each other**, especially with the statement `ask`. In the next chapter then, we will see how to **attach skills** to our species, giving them new attributes and actions. This section will be closed with a last lesson dealing with how **inheritence** works in GAML.

Chapter 23

The global species

We will start this chapter by studying a special species: the global species. In the global species you can define the attributes, actions and behaviors that describe the world agent. There is one unique world agent per simulation: it is this agent that is created when a user runs an experiment and that initializes the simulation through its **init** scope. The global species is a species like other and can be manipulated as them. In addition, the global species automatically inherits from several of built-in variables and actions. Note that a specificity of the global species is that all its attributes can be referred by all agents of the simulation.

Index

Declaration

A GAMA model contains a unique global section that defines the global species.

```
global {
    // definition of global attributes, actions, behaviours
}
```

global can use facets, such as the torus facet, to make the environment a torus or not (if it is a torus, all the agents going out of the environment will appear in the other side. If it's not, the agents won't be able to go out of the environment). By default, the environment is not a torus.

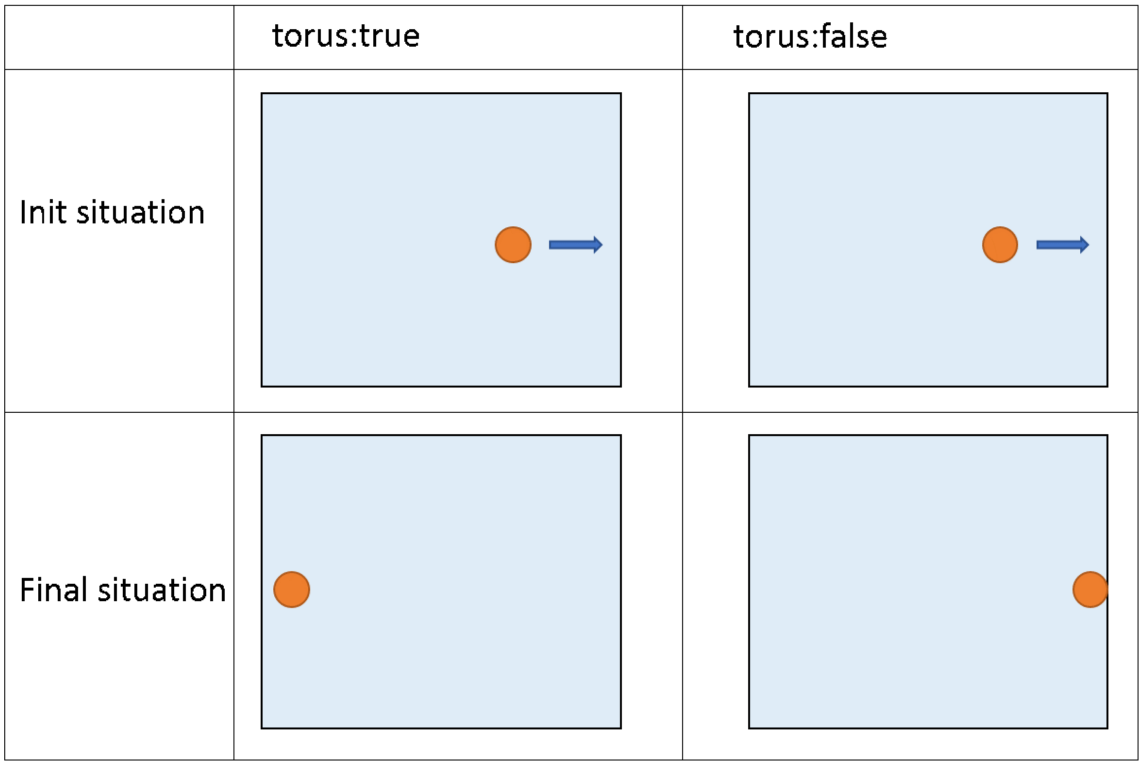


Figure 23.1: images/torus.png

```
global torus:true {
    // definition of global attributes, actions, behaviours
}
```

Other facets such as control or schedules are also available, but we will explain them later.

Directly in the global scope, you have to declare all your global attributes (can be seen as "static members" in Java or C++). To declare them, proceed exactly as for declaring basic variables. Those attributes are accessible wherever you want inside the species scope.

Environment size

In the global context, you have to define a size and a shape for your environment. In fact, an attribute already exists for the global species: it's called shape, and its type is a geometry. By default, shape is equal to a 100m*100m square. You can change the geometry of the shape by affecting another value:

```
geometry shape <- circle(50#mm);
geometry shape <- rectangle(10#m,20#m);
geometry shape <- polygon([{1°m,2°m},{3°m,50°cm},{3.4°m,60°dm}]);
```

nb: there are just examples. Try to avoid mixing dimensions! If no dimensions are specify, it'll be meter by default.

Built-in attributes

Some attributes exist by default for the global species. The attribute shape is one of them (refers to the shape of the environment). Here is the list of the other built-in attributes:

Like the other attributes of the global species, global built-in attributes can be accessed (and sometimes modified) by the world agent and every other agents in the model.

world

- represents the sole instance of the model species (i.e. the one defined in the `global` section). It is accessible from everywhere (including experiments) and gives access to built-in or user-defined global attributes and actions.

cycle

- integer, read-only, designates the (integer) number of executions of the simulation cycles. Note that the first cycle is the cycle with number 0.

To learn more about time, please read the recipe about dates.

step

- float, is the length, in model time, of an interval between two cycles, in seconds. Its default value is 1 (second). Each turn, the value of time is incremented by the value of step. The definition of step must be coherent with that of the agents' variables like speed. The use of time unit is particularly relevant for its definition.

To learn more about time, please read the recipe about dates.

```
global {
...
    float step <- 10°h;
...
}
```

time

- float, read-only, represents the current simulated time in seconds (the default unit). It is time in the model time. Begins at zero. Basically, we have: **time = cycle * step** .

```
global {
...
    int nb_minutes function: { int(time / 60)};
...
}
```

To learn more about time, please read the recipe about dates.

duration

- string, read-only, represents the value that is equal to the duration **in real machine time** of the last cycle.

total_duration

- string, read-only, represents the sum of duration since the beginning of the simulation.

average_duration

- string, read-only, represents the average of duration since the beginning of the simulation.

machine_time

- float, read-only, represents the current machine time in milliseconds.

agents

- list, read-only, returns a list of all the agents of the model that are considered as “active” (i.e. all the agents with behaviors, excluding the places). Note that obtaining this list can be quite time consuming, as the world has to go through all the species and get their agents before assembling the result. For instance, instead of writing something like:

```
ask agents of_species my_species {
...
}
```

one would prefer to write (which is much faster):

```
ask my_species {
...
}
```

Note that any agent has the `agents` attribute, representing the agents it contains. So to get all the agents of the simulation, we need to access the `agents` of the world using: `world.agents`.

Built-in Actions

The global species is provided with two specific actions.

halt

- stops the simulation.

```
global {
    ...
    reflex halting when: empty (agents) {
        do halt;
    }
}
```

pause

- pauses the simulation, which can then be continued by the user.

```
global {
    ...
    reflex toto when: time = 100 {
        do pause;
    }
}
```

The init statement

After declaring all the global attributes and defining your environment size, you can define an initial state (before launching the simulation). Here, you normally initialize your global variables, and you instantiate your species. We will see in the next session how to initialize a regular species. //: # (endConcept|global_species)

Chapter 24

Regular species

Regular species are composed of attributes, actions, reflex, aspect etc... They describes the behavior of our agents. You can instantiate as much as you want agents from a regular species, and you can define as much as you want different regular species. You can see a species as a "class" in OOP.

Index

Declaration

The regular species declaration starts with the keyword `species` followed by the name (or followed by the facet `name:`) :

```
species my_specie {
}
```

or:

```
species name:my_specie {
}
```

Directly in the "species" scope, you have to declare all your attributes (or "member" in OOP). You declare them exactly the way you declare basic variables. Those attributes are accessible wherever you want inside the species scope.

```
species my_specie {
    int variableA;
}
```

Built-in attributes

As for the global species, some attributes exist already by default in a regular species. Here is the list of built-in attributes:

- **name** (type: string) is used to name your agent. By default, the name is equal to the name of your species + an incremental number. This name is the one visible on the species inspector.
- **location** (type: point) is used to control the position of your agent. It refers to the center of the envelop of the shape associated to the agent.
- **shape** (type: geometry) is used to describe the geometry of your agent. If you want to use some intersection operator between agents for instance, it is this geometry that is computed (nb : it can be totally different from the aspect you want to display for your agent !). By default, the shape is a point.
- **host** (type: agent) is used when your agent is part of another agent. We will see this concept a bit further, in the topic multi-level architecture.

All those 4 built-in attributes can be accessed in both reading and writing very easily:

```
species my_species {
    init {
        name <- "custom_name";
        location <- {0,1};
        shape <- rectangle(5,1);
    }
}
```

All those built-in attributes are attributes of an agent (an instance of a species). Species has also their own attributes, which can be accessed with the following syntax (read only) :

```
name_of_your_species.attribute_you_want
```

Notice that the world agent is also an agent ! It has all the built-in attributes declared above. The world agent is defined inside the `global` scope. From the `global` scope then, you can for example access to the center of the envelop of the world shape :

```
global
{
    init {
        write location; // writes {50.0,50.0,0.0}
    }
}
```

Here is the list of those attributes:

- **name** (type: string) returns the name of your species
- **attributes** (type: list of string) returns the list of the names of the attributes of your species
- **population** (type: list) returns the list of agent that belong to it
- **subspecies** (type: list of string) returns the list of species that inherit directly from this species (we will talk about the concept of inheritance later)
- **parent** (type: species) returns its parent species if it belongs to the model, or `nil` otherwise (we will talk about the concept of inheritance later)

Built-in action

Some actions are define by default for a minimal agent. We already saw quickly the action `write`, used to display a message in the console. Another very useful built-in action is the action `die`, used to destroy an agent.

```
species my_species{
    reflex being_killed {
        do die;
    }
}
```

Here is the list of the other built-in actions which you can find in the documentation: debug, message, tell.

The init statement

After declaring all the attributes of your species, you can define an initial state (before launching the simulation). It can be seen as the "constructor of the class" in OOP.

```
species my_species {
    int variableA;
    init {
        variableA <- 5;
    }
}
```

The aspect statement

Inside each species, you can define one or several aspects. This scope allows you to define how you want your species to be represented in the simulation. Each aspect has a special name (so that they can be called from the experiment). Once again, you can name your aspect by using the facet `name:`, or simply by naming it just after the `aspect` keyword.

```
species my_species {
    aspect standard_aspect { // or "aspect name:standard_aspect"
    }
}
```

You can then define your aspect by using the statement `draw`. You can then choose a geometry for your aspect (facet `geometry`), a color (facet `color`), an image (facet `image`), a text (facet `text`)... We invite you to read the documentation about the draw statement to know more about.

```
species name:my_species {
    aspect name:standard_aspect {
        draw geometry:circle(1) color:#blue;
    }
}
```

In the experiment scope, you have to tell the program to display a particular species with a particular aspect (nb : you can also choose to display your species with several aspect in the same display).

```
experiment my_experiment type:gui
{
    output{
        display my_display {
            species my_species aspect:standard_aspect;
        }
    }
}
```

Now there is only one thing missing to display our agent: we have to instantiate them.

Instantiate an agent

As already said quickly in the last session, the instantiation of the agents is most often in the init scope of the global species (this is not mandatory of course. You can instantiate your agents from an action / behavior of any specie). Use the statement `create` to instantiate an agent. The facet species is used to specify which species you want to instantiate. The facet number is used to tell how many instantiation you want. The facet with is used to specify some default values for some attributes of your instance. For example, you can specify the location.

```
global{
    init{
        create species:my_species number:1 with:(location:{0,0},
   vA:8);
    }
}


species name:my_specie {
    int vA;
}
```

Here is an example of model that display an agent with a circle aspect in the center of the environment:

```
model display_one_agent

global{
    float worldDimension <- 50#m;
    geometry shape <- square(worldDimension);
    init{
        point center <- {(worldDimension/2),(worldDimension/2)};
        create species:my_species number:1 with:(location:center)
   ;
    }
}

species name:my_species {
    aspect name:standard_aspect {
        draw geometry:circle(1#m);
    }
}

experiment my_experiment type:gui
{
    output{
        display myDisplay {
            species my_species aspect:standard_aspect;
        }
    }
}
```

Chapter 25

Defining actions and behaviors

Both actions and behaviors can be seen as methods in OOP. They can be defined in any species.

Index

Action

Declare an action

An action is a function run by an instance of species. An action can return a value (in that case, the type of return has to be specify just before the name of the action), or not (in that case, you just have to put the keyword `action` before the name of the action).

```
species my_species {
    int action_with_return_value {
        // statements...
        return 1;
    }
    action action_without_return_value {
        // statements...
    }
}
```

Arguments can also be mandated in your action. You have to specify the type and the name of the argument:

```
action action_without_return_value (int argA, float argB) {
    // statements...
}
```

If you want to have some optional arguments in the list, you can give some by default values to turn them optional. Nb: it is better to define the optional arguments at the end of the list of argument.

```
action my_action (int argA, float argB <- 5.1, point argC <-
   {0,0}) {
    // statements...
}
```

Call an action

To call an action, you have to use the statement `do`. You can use the statement do different ways:

- With facets : after specifying the name of your action, you can specify the values of your arguments as if the name of your arguments were facets:

```
do my_action argA:5 argB:5.1;
```

- With parenthesis : after specifying the name of your action, you can specify the values of your arguments in the same order they were declared, between parenthesis:

```
do my_action (5,5.1);
```

We incite you to promote the second writing. To catch the returned value, you can also skip the do statement, and store the value directly in a temporary variable:

```
int var1 <- my_action(5,5.1);
```

Behavior

A behavior, or reflex, is an action which is called automatically at each time step by an agent.

```
reflex my_reflex {
    write ("Executing the inconditional reflex");
// statements...
}
```

With the facet when, this reflex is only executed when the boolean expression evaluates to true. It is a convenient way to specify the behavior of agents.

```
reflex my_reflex when:flip(0.5) {
    write ("Executing the conditional reflex");
// statements...
}
```

Reflex, unlike actions, cannot be called from another context. But a reflex can, of course, call actions.

Nb : Init is a special reflex, that occurs only when the agent is created.

Example

To practice a bit with those notions, we will build an easy example. Let's build a model with a species balloon that has 2 attributes: balloon_size (float) and balloon_color (rgb). Each balloon has a random position and color, his aspect is a sphere. Each step, a balloon has a probability to spawn in the environment. Once a balloon is created, its size is 10cm, and each step, the size increases by 1cm. Once the balloon size reaches 50cm, the balloon has a probability to burst. Once 10 balloons are destroyed, the simulation stops. The volume of each balloon is displayed in the balloon position.

Here is one of the multiple possible implementation:

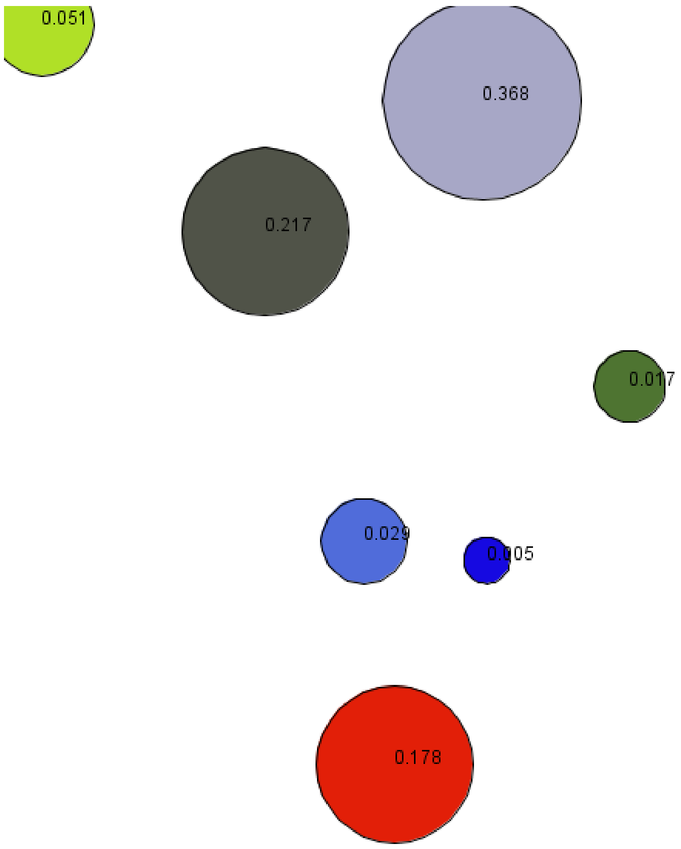


Figure 25.1: images/burst_the_baloon.png

```
model burst_the_baloon

global{
    float worldDimension <- 5#m;
    geometry shape <- square(worldDimension);
    int nbBaloonDead <- 0;

    reflex buildBaloon when:(flip(0.1)) {
        create species:balloon number:1;
    }

    reflex endSimulation when:nbBaloonDead>10 {
        do halt;
    }
}

species balloon {
    float balloon_size;
    rgb balloon_color;
    init {
        balloon_size <- 0.1;
        balloon_color <- rgb(rnd(255),rnd(255),rnd(255));
    }

    reflex balloon_grow {
        balloon_size <- balloon_size + 0.01;
        if (balloon_size > 0.5) {
            if (flip(0.2)) {
                do balloon_burst;
            }
        }
    }

    float balloon_volume (float diameter) {
        float exact_value <- 2/3*#pi*diameter^3;
        float round_value <- (round(exact_value*1000))/1000;
        return round_value;
    }

    action balloon_burst {
        write "the baloon is dead !";
        nbBaloonDead <- nbBaloonDead + 1;
        do die;
    }

    aspect balloon_aspect {
        draw circle(balloon_size) color:balloon_color;
        draw text:string(balloon_volume(balloon_size)) color:#
   black;
    }
}

experiment my_experiment type:gui
{
    output{
        display myDisplay {
            species balloon aspect:balloon_aspect;
        }
    }
}
```

Chapter 26

Interaction between agents

In this part, we will learn how interaction between agents works. We will also present you a bunch of operators useful for your modelling.

Index

The ask statement

The `ask` statement can be used in any reflex or action scope. It is used to specify the interaction between the instances of your species and the other agents. You only have to specify the species of the agents you want to interact with. Here are the different ways of calling the ask statement:

- If you want to interact with one particular agent (for example, defined as an attribute of your species):

```
species my_species {
    agent target;
    reflex update {
        ask target {
            // statements
        }
    }
}
```

- If you want to interact with a group of agents:

```
species my_species {
    list<agent> targets;
    reflex update {
        ask targets {
            // statements
        }
    }
}
```

- If you want to interact with agents, as if they were instance of a certain species (can raise an error if it's not the case!):

```
species my_species {
    list<agent> targets;
    reflex update {
        ask targets as:my_species {
            // statements
        }
    }
}
```

- If you want to interact with all the agent of a species:

```
species my_species {
    list<agent> targets;
    reflex update {
        ask other_species {
            // statements
        }
    }
}

species other_species {
}
```

Note that you can use the attribute *population* of `species` if you find it more explicit:

```
ask other_species.population
```

- If you want to interact with all the agent of a particular species from a list of agents (for example, using the global variable "agents"):

```
species my_specie {
    reflex update {
        ask species of_species my_specie {
            // statements
        }
    }
}
```

Pseudo variables

Once you are in the ask scope, you can use some pseudo variables to refer to the receiver agent (the one specify just after the ask statement) or the transmitter agent (the agent which is asking). We use the pseudo variable `self` to refer to the receiver agent, and the pseudo variable `myself` to refer to the transmitter agent. The pseudo variable `self` can be omitted when calling actions or attributes.

```
species speciesA {
    init {
        name <- "speciesA";
    }
    reflex update {
        ask speciesB {
write name; // output : "speciesB"
write self.name; // output : "speciesB"
            write myself.name; // output : "speciesA"
        }
    }
}

species speciesB {
    init {
        name <- "speciesB";
    }
}
```

Now, if we introduce a third species, we can write an `ask` statement inside another.

```
species speciesA {
    init {
        name <- "speciesA";
    }
    reflex update {
        ask speciesB {
            write self.name; // output : "speciesB"
            write myself.name; // output : "speciesA"
            ask speciesC {
                write self.name; // output : "speciesC"
                write myself.name; // output : "speciesB"
            }
        }
    }
}

species speciesB {
    init {
        name <- "speciesB";
    }
}

species speciesC {
    init {
        name <- "speciesC";
    }
}
```

Nb: try to avoid multiple imbrications of ask statements. Most of the time, there is another way to do the same thing.

Some useful interaction operators

The operator `at_distance` can be used to know the list of agents that are in a certain distance from another agent.

```
species my_species {
    reflex update {
        list<agent> neighbours <- agents at_distance(5);
        // neighbours contains the list of all the agents located
    at a distance <= 5 from the caller agent.
    }
}
```

The operator `closest_to` returns the closest agent of a position among a container.

```
species my_species {
    reflex update {
        agent agentA <- agents closest_to(self);
        // agentA contains the closest agent from the caller
   agent.
        agent agentB <- other_specie closest_to({2,3});
        // agentB contains the closest instance of other_specie
   from the location {2,3}.
    }
}

species other_specie {
}
```

Example

To practice those notions, here is a short basic example. Let's build a model with a fix number of agents with a circle shape. They can move randomly on the environment, and when they are close enough from another agent, a line is displayed between them. This line is destroyed when the distance between the two agents is too important. Hint: use the operator `polyline` to construct a line. List the points between angle brackets `[]`.

Here is one example of implementation:

```
model connect_the_neighbours

global{
    float speed <- 0.2;
    float distance_to_intercept <- 10.0;
    int number_of_circle <- 100;
    init {
        create my_species number:number_of_circle;
    }
}

species my_species {
    reflex move {
        location <- {location.x+rnd(-speed,speed),location.y+rnd
    (-speed,speed)};
    }
    aspect default {
        draw circle(1);
        ask my_species at_distance(distance_to_intercept) {
            draw polyline([self.location,myself.location]) color
    :#black;
        }
    }
}

experiment my_experiment type:gui
{
    output{
        display myDisplay {
            species my_species aspect:default;
        }
```

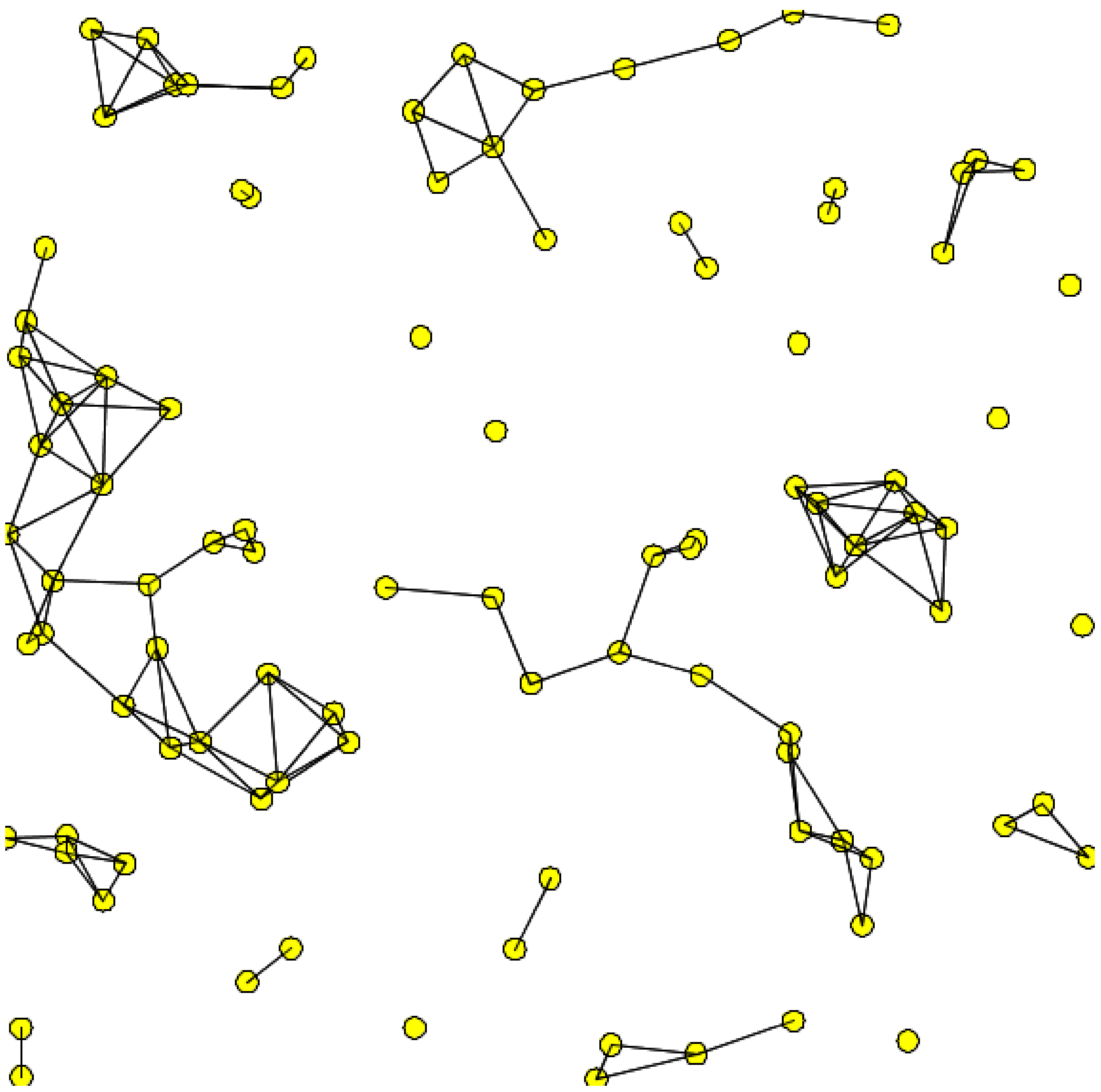


Figure 26.1: images/connect_the_neighbours.png

```
    }
}
```

Chapter 27

Attaching Skills

GAMA allows to attach skills to agents through the facet `skills`. Skills are built-in modules that provide a set of related built-in attributes and built-in actions (in addition to those already proposed by GAMA) to the species that declare them.

Index

Skills

A declaration of skill is done by filling the `skills` facet in the species definition:

```
species my_species skills: [skill1,skill2] {
}
```

A very useful and common skill is the `moving` skill.

```
species my_species skills: [moving] {
}
```

Once your species has the moving skill, it earns automatically the following attributes: `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `follow`, `wander` and `wander_3D`.

Attributes:

- **`speed`** (float) designs the speed of the agent, in m/s.
- **`heading`** (int) designs the heading of an agent in degrees, which means that is the maximum angle the agent can turn around each step.
- **`destination`** (point) is the updated destination of the agent, with respect to its speed and heading. It's a read-only attribute, you can't change its value.

Actions:

`follow`

moves the agent along a given path passed in the arguments.

- returns: path
- **`speed`** (float): the speed to use for this move (replaces the current value of speed)
- **`path`** (path): a path to be followed.
- **`move_weights`** (map): Weights used for the moving.
- **`return_path`** (boolean): if true, return the path followed (by default: false)

`goto`

moves the agent towards the target passed in the arguments.

- returns: path
- **`target`** (agent,point,geometry): the location or entity towards which to move.

- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `on` (graph): graph that restrains this move
- `recompute_path` (boolean): if false, the path is not recompute even if the graph is modified (by default: true)
- `return_path` (boolean): if true, return the path followed (by default: false)
- `move_weights` (map): Weights used for the moving.

`move`

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `heading` (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- `bounds` (geometry,agent): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

`wander`

Moves the agent towards a random location at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- returns: void
- `speed` (float): the speed to use for this move (replaces the current value of speed)

- `amplitude` (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- `bounds` (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

`wander_3D`

Moves the agent towards a random location (3D point) at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `amplitude` (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- `z_max` (int): the maximum altitude (z) the geometry can reach
- `bounds` (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

Other skills

A lot of other skills are available. Some of them can be built in skills, integrated by default in GAMA, other are linked to additional plugins.

This is the list of skills: `Advanced_driving`, `communication`, `driving`, `GAMASQL`, `graphic`, `grid`, `MDXSKILL`, `moving`, `moving3D`, `physical3D`, `skill_road`, `skill_road`, `skill_road_node`, `SQLSKILL`

Example

We can now build a model using the skill moving. Let's design 2 species, one is "species_red", the other is "species_green". Species_green agents are moving randomly with a certain speed and a certain heading. Species_red agents wait for a species_green agent to be in a certain range of distance. Once it is the case, the agent move toward the species_green agent. A line link the red_species agent and its target.

Here is an example of implementation:

```
model green_and_red_species

global{
    float distance_to_intercept <- 10.0;
    int number_of_green_species <- 50;
    int number_of_red_species <- 50;
    init {
        create speciesA number:number_of_green_species;
        create speciesB number:number_of_red_species;
    }
}

species speciesA skills:[moving] {
    init {
        speed <- 1.0;
    }
    reflex move {
        do wander amplitude:90;
    }
    aspect default {
        draw circle(1) color:#green;
    }
}

species speciesB skills:[moving] {
    speciesA target;
    init {
        speed <- 0.0;
        heading <- 90;
    }
    reflex search_target when:target=nil {
        ask speciesA at_distance(distance_to_intercept) {
            myself.target <- self;
        }
    }
    reflex follow when:target!=nil {
```

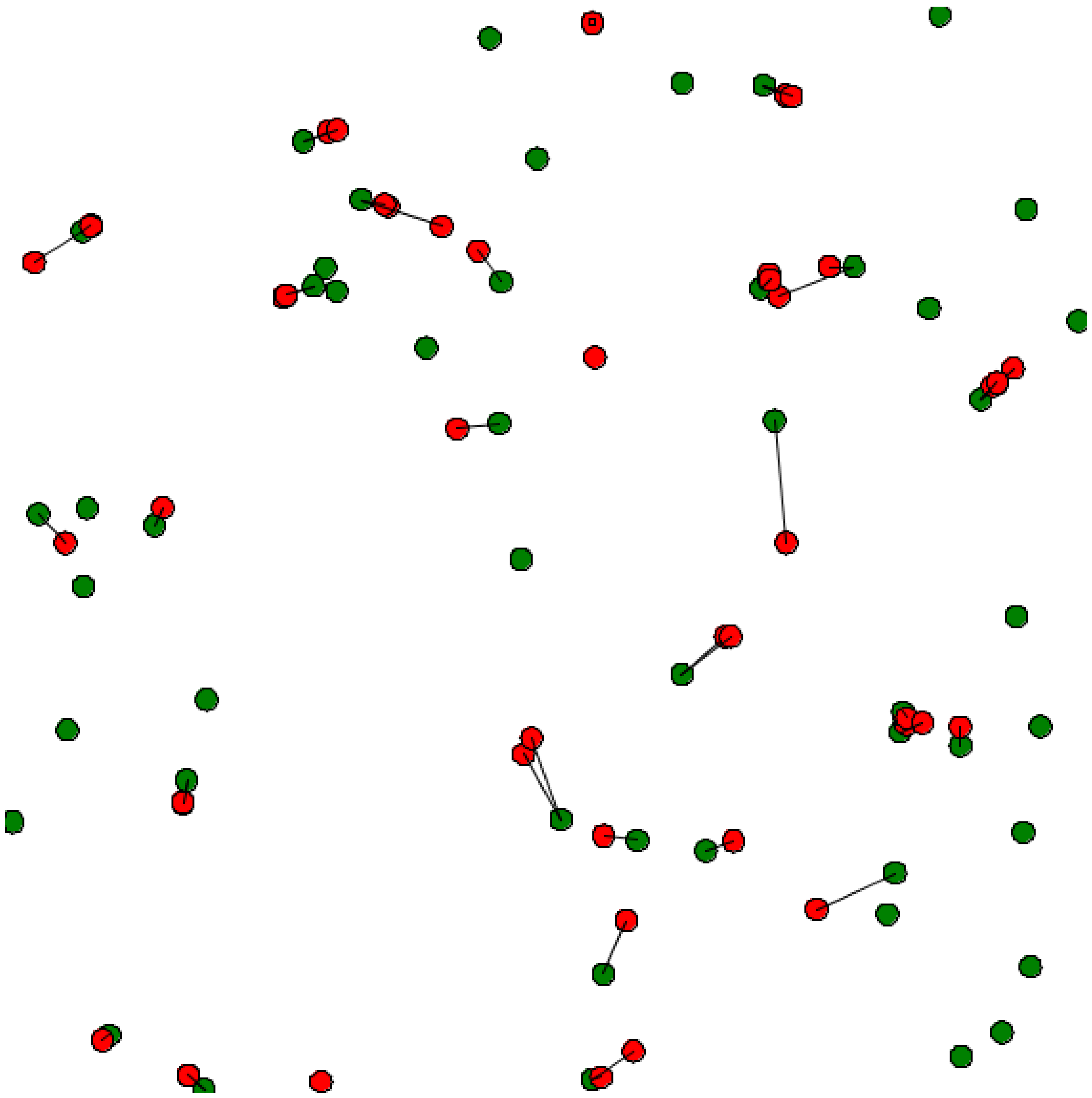


Figure 27.1: images/green_and_red_species.png

```
        speed <- 0.8;
        do goto target:target;
    }
    aspect default {
        draw circle(1) color:#red;
        if (target!=nil) {
            draw polyline([self.location,target.location]) color
    :#black;
        }
    }
}

experiment my_experiment type:gui
{
    output{
        display myDisplay {
            species speciesA aspect:default;
            species speciesB aspect:default;
        }
    }
}
```

Chapter 28

Inheritance

As for multiple programming language, inheritance can be used in GAML. It is used to structure better your code, when you have some complex models.

Index

Mother species / child species

To make a species inherit from a mother species, you have to add the facet `parent`, and specify the mother species.

```
species mother_species {
}

species child_species parent:mother_species {
}
```

Thus, all the attributes, actions and reflex of the mother species are inherited to the child species.

```
species mother_species {
    int attribute_A;
    action action_A {}
}

species child_species parent:mother_species {
    init {
        attribute_A <- 5;
        do action_A;
    }
}
```

If the mother species has a particular skill, its children will inherit all the attributes and actions.

```
species mother_species skills:[moving] {
}

species child_species parent:mother_species {
    init {
        speed <- 2.0;
    }
    reflex update {
        do wander;
    }
}
```

You can redefine an action or a reflex by declaring an action or a reflex with the same name.

Virtual action

You have also the possibility to declare a virtual action in the mother species, which means an action without implementation, by using the facet `virtual`:

```
action virtual_action virtual:true;
```

When you declare an action as virtual in a species, this species becomes abstract, which means you cannot instantiate agent from it. All the children of this species has to implement this virtual action.

```
species virtual_mother_species {
    action my_action virtual:true;
}

species child_species parent:virtual_mother_species {
    action my_action {
        // some statements
    }
}
```

Get all the subspecies from a species

If you declare a "mother" species, you create a "child" agent, then "mother" will return the population of agents "mother" and **not** the population of agents "child", as it is shown in the following example :

```
global
{
    init {
        create child number:2;
        create mother number:1;
    }
    reflex update {
        write length(mother); // will write 1 and not 3
    }
}

species mother {}

species child parent:mother {}
```

We reminds you that "subspecies" is a built-in attribute of the agent. Using this attribute, you can easily get all the subspecies agents of the mother species by writing the following gaml function :

```
global
{
    init {
        create child number:2;
        create mother number:1;
    }
    reflex update {
        write length(get_all_instances(mother)); // will write 3
   (1+2)
    }
    list<agent> get_all_instances(species<agent> spec) {
        return spec.population +  spec.subspecies accumulate (
   get_all_instances(each));
    }
}

species mother {}

species child parent:mother {}
```

Defining Advanced Species

In the previous chapter, we saw how to declare and manipulate **regular species** and the **global species** (as a reminder, the instance of the **global species** is the **world agent**).

We will now see that GAMA provides you the possibility to declare some special species, such as **grids** or **graphs**, with their own built-in attributes and their own built-in actions. We will also see how to declare **mirror species**, which is a "copy" of a regular species, in order to give it an other representation. Finally, we will learn how to represent several agents through one unique agent, with **multi-level architecture**.

Chapter 29

Grid Species

A grid is a particular species of agents. Indeed, a grid is a set of agents that share a grid topology (until now, we only saw species with continuous topology). As other agents, a grid species can have attributes, attributes, behaviors, aspects However, contrary to regular species, grid agents are created automatically at the beginning of the simulation. It is thus not necessary to use the create statement to create them. Moreover, in addition to classic built-in variables, grid a provided with a set of additional built-in variables.

Index

Declaration

Instead of using the `species` keyword, use the keyword `grid` to declare a grid species. The grid species has exactly the same facets of the regular species, plus some others. To declare a grid, you have to specify the number of columns and rows first. You can do it two different ways:

Using the two facets `width`: and `height`: to fix the number of cells (the size of each cells will be determined thanks to the environment dimension).

```
grid my_grid width:8 height:10 {
// my_grid has 8 columns and 10 rows
}
```

Using the two facets `cell_width`: and `cell_height`: to fix the size of each cells (the number cells will be determined thanks to the environment dimension).

```
grid my_grid cell_width:3 cell_height:2 {
// my_grid has cells with dimension 3m width by 2m height
}
```

By default, a grid is composed by 100 rows and 100 columns.

Another facet exists for grid only, very useful. It is the `neighbors` facet, used to determine how many neighbors has each cell. You can choose among 3 values: 4 (Von Neumann), 6 (hexagon) or 8 (Moore).

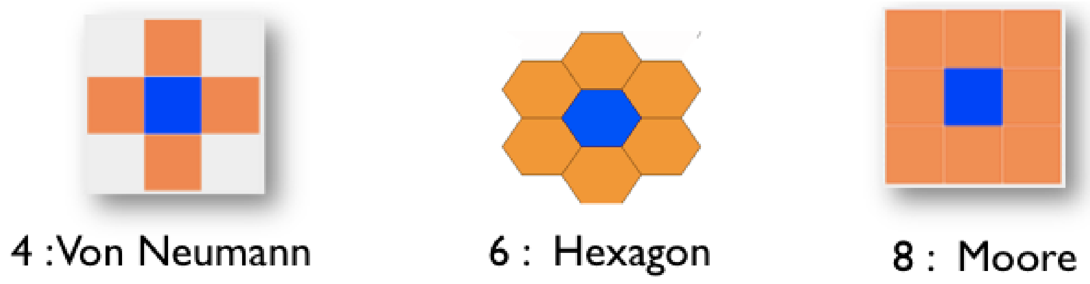


Figure 29.1: images/grid_neighbors.png

A grid can also be provided with specific facets that allows to optimize the computation time and the memory space, such as `use_regular_agents`, `use_indivitual_shapes` and `use_neighbours_cache`. Please refer to the GAML Reference for more explanation about those particular facets.

Built-in attributes

grid_x

This variable stores the column index of a cell.

```
   grid cell width: 10 height: 10 neighbors: 4 {
         init {
               write "my column index is:" + grid_x;
         }
   }
```

grid_y

This variable stores the row index of a cell.

```
   grid cell width: 10 height: 10 neighbors: 4 {
         init {
               write "my row index is:" + grid_y;
         }
   }
```

agents

return the set of agents located inside the cell. Note the use of this variable is deprecated. It is preferable to use the `inside` operator: //: # (keyword|operator_inside)

```
grid cell width: 10 height: 10 neighbors: 4 {
         list<bug> bugs_inside -> {bug inside self};
   }
```

color

The **color** built-in variable is used by the optimized grid display. Indeed, it is possible to use for grid agents an optimized aspect by using in a display the **grid** keyword. In this case, the grid will be displayed using the color defined by the **color** variable. The border of the cells can be displayed with a specific color by using the **lines** facet.

Here an example of the display of a grid species named **cell** with black border.

```
experiment main_xp type: gui{
     output {
        display map {
            grid cell lines: rgb("black") ;
        }
    }
}
```

neighbors

The **neighbors** built-in variable returns the list of cells at a distance of 1.

```
grid my_grid {
  reflex writeNeighbors {
    write neighbors;
  }
}
```

grid_value

The **grid_value** built-in variable is used when initializing a grid from grid file (see later). It is also used for the 3D representation of DEM.

Access to a cell

there are several ways to access to a specific cell:

- by a location: by casting a location to a cell (the unity (`#m`, `#cm`, etc...) is defined when you choose your environment size, in the global species.

```
   global {
         init {
             write "cell at {57.5, 45} :" + cell({57.5, 45});
         }
   }

   grid cell width: 10 height: 10 neighbors: 4 {
   }
```

- by the row and column indexes: like matrix, it is possible to directly access to a cell from its indexes

```
global {
    init {
        write "cell [5,8] :" + cell[5,8];
    }
}
grid cell width: 10 height: 10 neighbors: 4 {
}
```

The operator `grid_at` also exists to get a particular cell. You just have to specify the index of the cell you want (in x and y):

```
global {
    init {
        agent cellAgent <- cell grid_at {5,8};
        write "cell [5,8] :" + cellAgent;
    }
}
grid cell width: 10 height: 10 neighbors: 4 {
}
```

Display Grid

You can easily display your grid in your experiment as followed :

```
experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            grid MyGrid;
        }
    }
}
```

The grid will be displayed, using the color you defined for each cell (with the "color" built-in attribute). You can also show border of each cell by using the facet "line:" and choosing a rgb color:

```
grid MyGrid line:#black;
```

An other way to display a grid will be to define an aspect in your grid agent (the same way as for a regular species), and define your grid as a regular species then in your experiment, choosing your aspect :

```
grid MyGrid {
    aspect firstAspect {
        draw square(1);
    }
    aspect secondAspect {
        draw circle(1);
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            species MyGrid aspect:firstAspect;
        }
    }
}
```

Beware : don't use this second display when you have large grids : it's much slower.

Grid from a matrix

An easy way to load some values in a grid is to use matrix data. A `matrix` is a type of container (we invite you to learn some more about this useful type here). Once you have declared your matrix, you can set the values of your cells using the `ask` statement :

```
global {
  init {
    matrix data <- matrix([[0,1,1],[1,2,0]]);
    ask cell {
      grid_value <- float(data[grid_x, grid_y]);
    }
  }
}
```

Declaring larger matrix in GAML can be boring as you can imagine. You can load your matrix directly from a csv file with the operator `matrix` (used for the contruction of the matrix).

```
file my_file <- csv_file("path/file.csv","separator");
matrix my_matrix <- matrix(my_file);
```

You can try to read the following csv :

```
0,0,0,0,0,0,0,0,0,0,0
0,0,0,1,1,1,1,1,0,0,0
0,0,1,1,0,0,0,1,1,0,0
0,1,1,0,0,0,0,0,0,0,0
0,1,1,0,0,1,1,1,1,0,0
0,0,1,1,0,0,1,1,1,0,0
0,0,0,1,1,1,1,0,1,0,0
0,0,0,0,0,0,0,0,0,0,0
```

With the following model :

```
model import_csv

global {
  file my_csv_file <- csv_file("../includes/test.csv",",");
  init {
    matrix data <- matrix(my_csv_file);
    ask my_gama_grid {
      grid_value <- float(data[grid_x,grid_y]);
      write data[grid_x,grid_y];
    }
  }
}

grid my_gama_grid width: 11 height: 8 {
  reflex update_color {
    write grid_value;
    color <- (grid_value = 1) ? #blue : #white;
  }
}

experiment main type: gui{
  output {
    display display_grid {
      grid my_gama_grid;
    }
  }
}
```

For more complicated models, you can read some other files, such as ASCII files (asc), DEM files…

Example

To practice a bit those notions, we will build a quick model. A "regular" species will move randomly on the environment. A grid is displayed, and its cells becomes red when an instance of the regular species is waking inside this cell, and yellow when the regular agent is in the surrounding of this cell. If no regular agent is on the surrounding, the cell turns green.

Here is an example of implementation:

```
model my_grid_model

global{
    float max_range <- 5.0;
    int number_of_agents <- 5;
    init {
        create my_species number:number_of_agents;
    }
    reflex update {
        ask my_species {
            do wander amplitude:180;
            ask my_grid at_distance(max_range)
            {
                if(self overlaps myself)
                {
                    self.color_value <- 2;
                }
                else if (self.color_value != 2)
                {
                    self.color_value <- 1;
                }
            }
        }
```

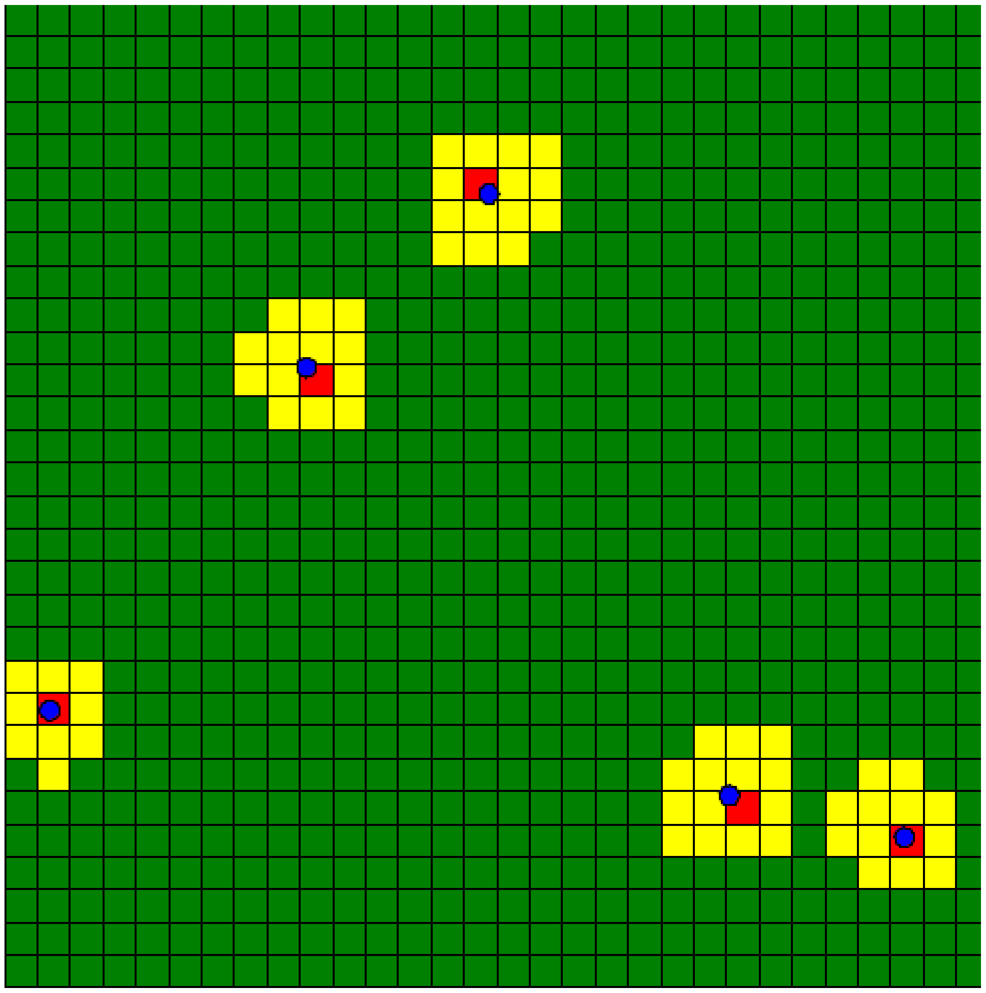


Figure 29.2: images/my_grid_model.png

```
        ask my_grid {
            do update_color;
        }
    }
}

species my_species skills:[moving] {
    float speed <- 2.0;
    aspect default {
        draw circle(1) color:#blue;
    }
}

grid my_grid width:30 height:30 {
    int color_value <- 0;
    action update_color {
        if (color_value = 0) {
            color <- #green;
        }
        else if (color_value = 1) {
            color <- #yellow;
        }
        else if (color_value = 2) {
            color <- #red;
        }
        color_value <- 0;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            grid my_grid lines:#black;
            species my_species aspect:default;
        }
    }
}
```

Chapter 30

Graph Species

Using a graph species enables to easily show interaction between agents of a same species. This kind of species is particularly useful when trying to show the interaction (especially the non-spatial one) that exist between agents.

Index

Declaration

Declare a graph with handmade agents

To instantiate this graph species, several steps must be followed. First the graph species must inherit from the abstract species `graph_node`, then the method `related_to` must be redefined and finally an auxiliary species that inherits from `base_edge` used to represent the edges of the generated graph must be declared. A graph node is an abstract species that must redefine one method called `related_to`.

```
species graph_agent parent: graph_node edge_species: edge_agent{
  bool related_to(graph_agent other){
    return true;
  }
}

species edge_agent parent: base_edge {
}
```

The method `related_to` returns a boolean, and take the agents from the current species in argument. If the method returns true, the two agents (the current instance and the one as argument) will be linked.

```
global{
    int number_of_agents <- 5;
    init {
        create graph_agent number:number_of_agents;
    }
}

species graph_agent parent: graph_node edge_species: edge_agent{
  bool related_to(graph_agent other){
    return true;
  }
  aspect base {
    draw circle(1) color:#green;
  }
}

species edge_agent parent: base_edge {
    aspect base {
```

```
    draw shape color:#blue;
  }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species graph_agent aspect:base;
            species edge_agent aspect:base;
        }
    }
}
```

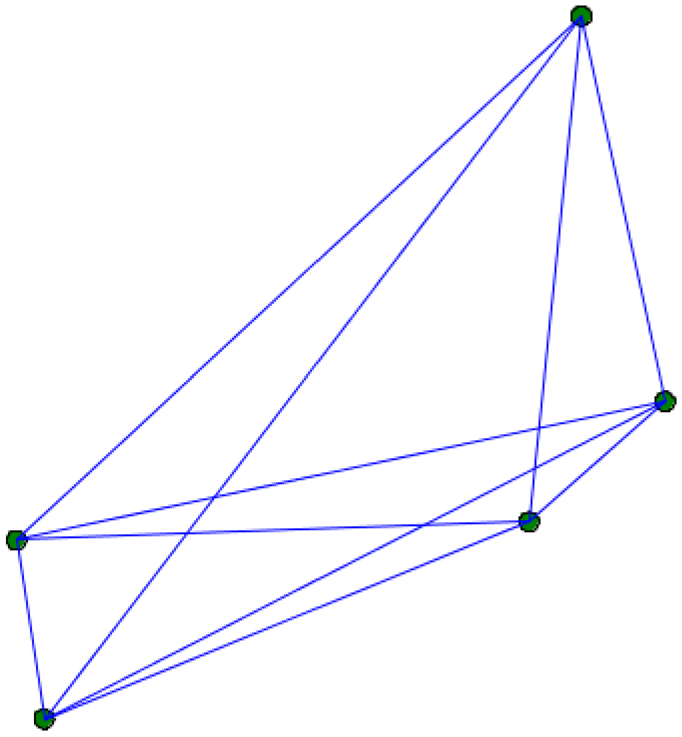


Figure 30.1: graph_related_to.png

You can for example link 2 agents when they are closer than a certain distance. Beware: The topology used in graph species is the graph topology, and not the continuous topology. You can force the use of the continuous topology with the action `using` as follow:

```
bool related_to(graph_agent other){
    using topology:topology(world) {
        return (self.location distance_to other.location < 20);
    }
}
```

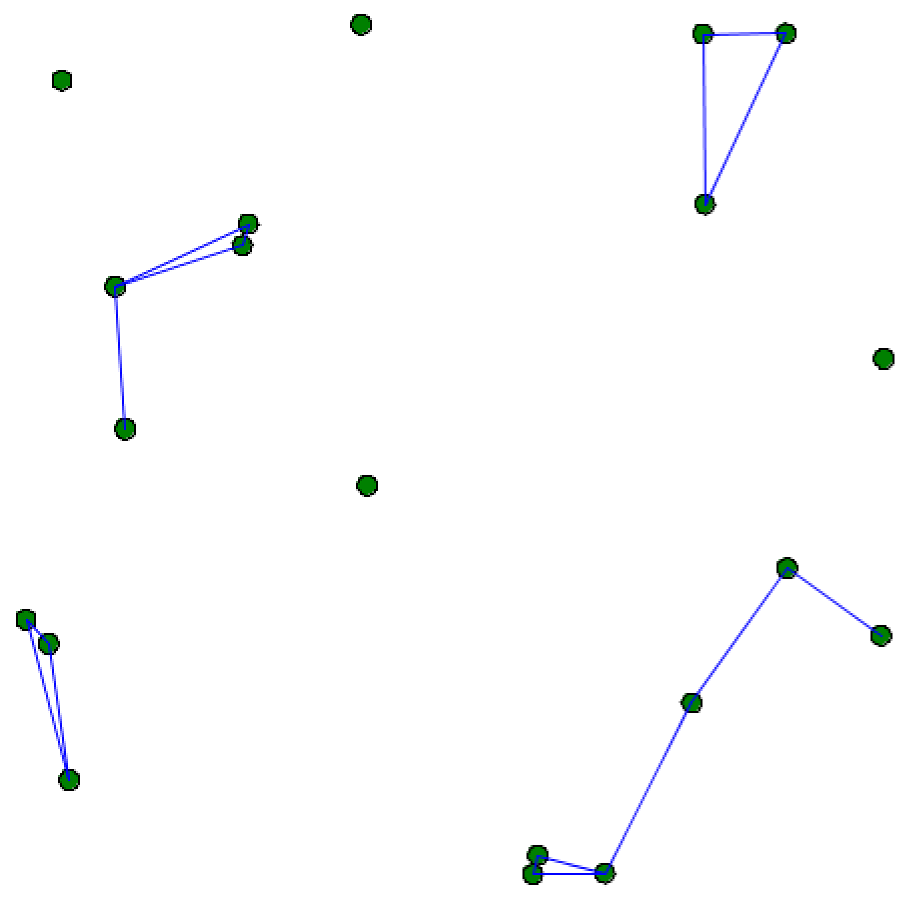


Figure 30.2: graph_related_to2.png

The abstract mother species "graph_node" has an attribute "my_graph", with the type "graph". The graph type represent a graph composed of vertices linked with edges. This type has built-in attributes such as `edges` (the list of all the edges agents), or `vertices` (the list of all the vertices agents).

Declare a graph by using an geometry file

In most cases, you will have to construct a graph from an existing file (example: a "shp" file). In that case, you will have to first instantiate a species from the shape file (with the `create` statement, using the facet `from`). When, you will have to extract a graph from the agent, using the operator `as_edge_graph`.

```
model load_shape_file

global {
    file roads_shapefile <- file("../includes/road.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;

    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

experiment main_experiment type:gui{
    output {
        display map {
            species road aspect:geom;
        }
    }
}
```

Declare a graph with nodes and edges

Another way to create a graph is building it manually nodes by nodes, and then edges by edges, without using agent structures. Use the `add_node` operator and the `add_edge` operator to do so. Here is an example of how to do:

```
add point(0.0,0.0) to:nodes;
add point(90.0,90.0) to:nodes;
add point(20.0,20.0) to:nodes;
add point(40.0,50.0) to:nodes;
add point(100.0,0.0) to:nodes;

loop nod over:nodes {
    my_graph <- my_graph add_node(nod);
}

my_graph <- my_graph add_edge (nodes at 0::nodes at 2);
my_graph <- my_graph add_edge (nodes at 2::nodes at 3);
my_graph <- my_graph add_edge (nodes at 3::nodes at 1);
my_graph <- my_graph add_edge (nodes at 0::nodes at 4);
my_graph <- my_graph add_edge (nodes at 4::nodes at 1);
```

Using this solution, my_graph can have two types: it can be an a-spatial graph, or a spatial graph. The spatial graph will have a proper geometry, with segments that follow the position of your graph (you can access to the segments by using the built-in "segments"). The a-spatial graph will not have any shape.

```
global
{
    graph my_spatial_graph<-spatial_graph([]);
    graph my_aspatial_graph<-graph([]);

    init {
        point node1 <- {0.0,0.0};
        point node2 <- {10.0,10.0};
        my_spatial_graph <- my_spatial_graph add_node(node1);
        my_spatial_graph <- my_spatial_graph add_node(node2);
        my_spatial_graph <- my_spatial_graph add_edge(node1::
   node2);
        write my_spatial_graph.edges;
        // the output is [polyline
   ([{0.0,0.0,0.0},{10.0,10.0,0.0}])]
        my_aspatial_graph <- my_aspatial_graph add_node(node1);
        my_aspatial_graph <- my_aspatial_graph add_node(node2);
        my_aspatial_graph <- my_aspatial_graph add_edge(node1::
   node2);
```

```
        write my_aspatial_graph.edges;
        // the output is [{0.0,0.0,0.0}::{10.0,10.0,0.0}]
    }
}
```

Useful operators with graph

Knowing the degree of a node

The operator `degree_of` returns the number of edge attached to a node. To use it, you have to specify a graph (on the left side of the operator), and a node (on the right side of the operator).

The following code (to put inside the node species) displays the number of edges attached to each node:

```
aspect base
{
    draw text:string(my_graph degree_of node(5)) color:# black;
    status <- 0;
}
```

Get the neighbors of a node

To get the list of neighbors of a node, you should use the neighbors_of operator. On the left side of the operator, specify the graph you are using, and on the right side, specify the node. The operator returns the list of nodes located at a distance inferior or equal to 1, considering the graph topology.

```
species graph_agent parent: graph_node edge_species: edge_agent
{
  list<graph_agent> list_neighbors <- list<graph_agent>(my_graph
   neighbors_of (self));
}
```

Here is an example of model using those two previous concepts (a random node is chosen each step, displayed in red, and his neighbors are displayed in yellow):

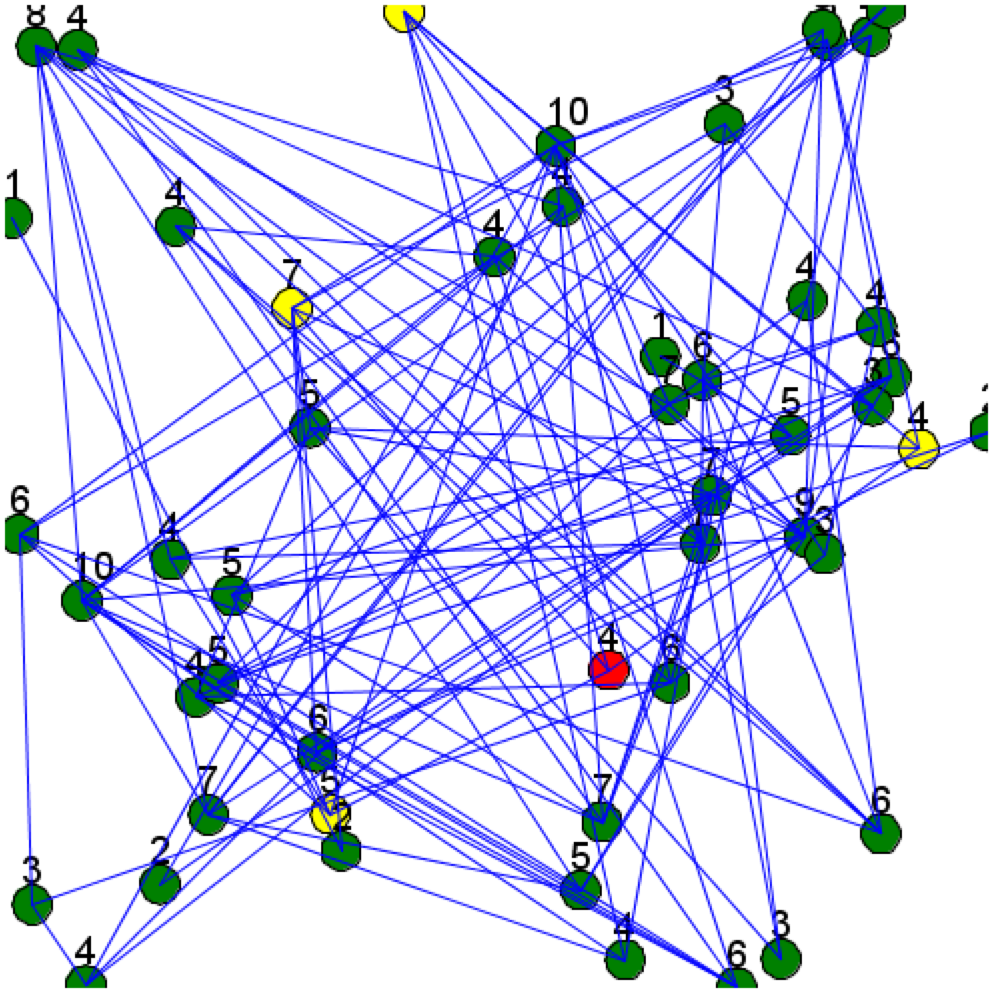


Figure 30.3: graph_model.png

```
model graph_model

global
{
    int number_of_agents <- 50;
    init
    {
        create graph_agent number: number_of_agents;
    }

    reflex update {
        ask graph_agent(one_of(graph_agent)) {
            status <- 2;
            do update_neighbors;
        }
    }
}

species graph_agent parent: graph_node edge_species: edge_agent
{
    int status <- 0;
    list<int> list_connected_index;

    init {
        int i<-0;
        loop over:graph_agent {
            if (flip(0.1)) {
                add i to:list_connected_index;
            }
            i <- i+1;
        }
    }

    bool related_to(graph_agent other){
        if (list_connected_index contains (graph_agent index_of
   other)) {
            return true;
        }
        return false;
    }
    action update_neighbors {

        list<graph_agent> list_neighbors <- list<graph_agent>(
   my_graph neighbors_of (self));

        loop neighb over:list_neighbors {
            neighb.status <- 1;
        }
    }

    aspect base
    {
        if (status = 0) {
            draw circle(2) color: # green;
        }
        else if (status = 1) {
            draw circle(2) color: # yellow;
        }
        else if (status = 2) {
            draw circle(2) color: # red;
        }
        draw text:string(my_graph degree_of self) color:# black
   size:4 at:point(self.location.x-1,self.location.y-2);
        status <- 0;
    }
}

species edge_agent parent: base_edge
{
    aspect base
    {
        draw shape color: # blue;
    }
}

experiment MyExperiment type: gui
{
    output
    {
        display MyDisplay type: java2D
        {
            species graph_agent aspect: base;
            species edge_agent aspect: base;
        }
    }
}
```

Compute the shortest path

To compute the shortest path to go from a point to another, pick a source and a destination among the vertices you have for your graph. Store those values as point type.

```
point source;
point destination;
source <- point(one_of(my_graph.vertices));
destination <- point(one_of(my_graph.vertices));
```

Then, you can use the operator `path_between` to return the shortest path. To use this action, you have to give the graph, then the source point, and the destination point. This action returns a path type.

```
path shortest_path;
shortest_path <- path_between (my_graph, source,destination);
```

Another operator exists, `paths_between`, that returns a list of shortest paths between two points. Please read the documentation to learn more about this operator.

Here is an example of code that show the shortest path between two points of a graph:

```
model graph_model

global
{
    int number_of_agents <- 50;
    point source;
    point target;
    graph my_graph;
    path shortest_path;

    init
```

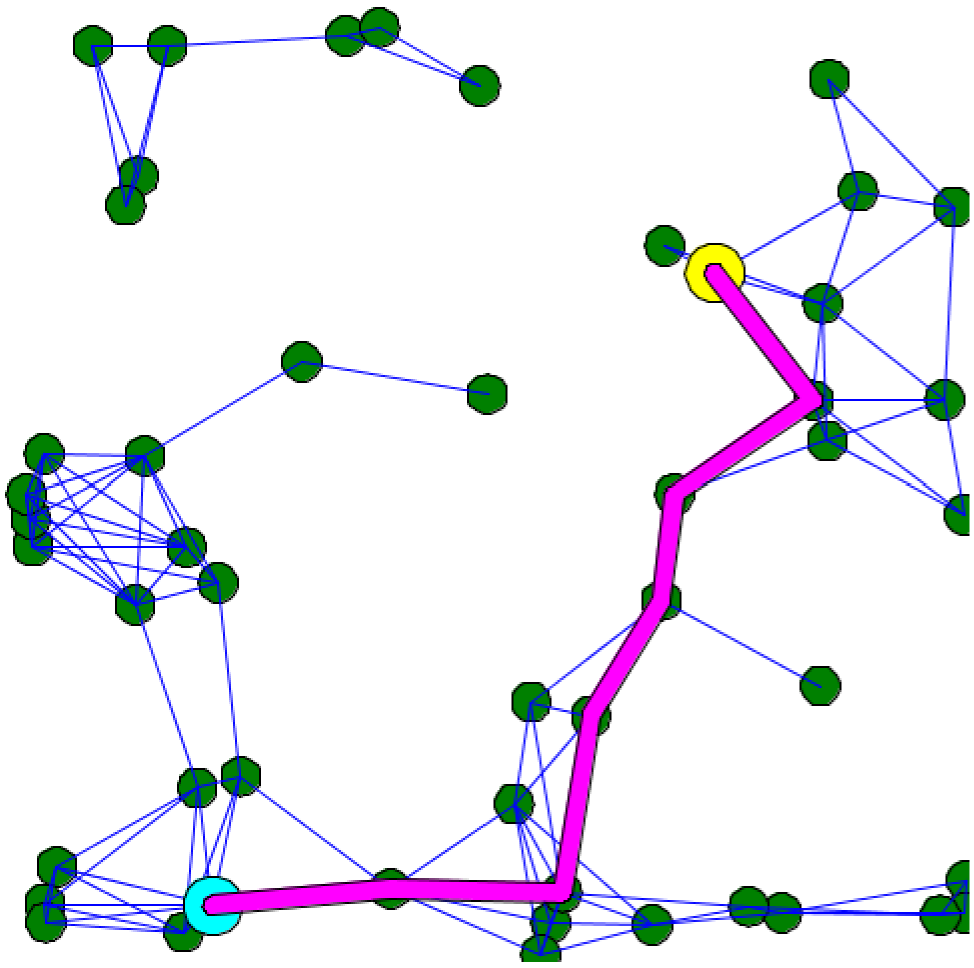


Figure 30.4: shortest_path.png

```
    {
        create graph_agent number: number_of_agents;
    }

    reflex pick_two_points {
        if (my_graph=nil) {
            ask graph_agent {
                myself.my_graph <- self.my_graph;
                break;
            }
        }
        shortest_path <- nil;
        loop while:shortest_path=nil {
            source <- point(one_of(my_graph.vertices));
            target <- point(one_of(my_graph.vertices));
            if (source != target) {
                shortest_path <- path_between (my_graph, source,
   target);
            }
        }
    }
}

species graph_agent parent: graph_node edge_species: edge_agent
{
    list<int> list_connected_index;

    init {
        int i<-0;
        loop over:graph_agent {
            if (flip(0.1)) {
                add i to:list_connected_index;
            }
            i <- i+1;
        }
    }

    bool related_to(graph_agent other) {
        using topology:topology(world) {
            return (self.location distance_to other.location <
   20);
```

```
        }
    }

    aspect base {
        draw circle(2) color: # green;
    }
}

species edge_agent parent: base_edge
{
    aspect base {
        draw shape color: # blue;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species graph_agent aspect: base;
            species edge_agent aspect: base;
            graphics "shortest path" {
                if (shortest_path != nil) {
                    draw circle(3) at: source color: #yellow;
                    draw circle(3) at: target color: #cyan;
                    draw (shortest_path.shape+1) color: #magenta;
                }
            }
        }
    }
}
```

Control the weight in graph

You can add a map of weight for the edges that compose the graph. Use the operator `with_weights` to put weights in your graph. The graph has to be on the left side of the operator, and the map has to be on the right side. In the map, you have to put edges as key, and the weight for that edge as value. One common use is to put the distance as weight:

```
my_graph <- my_graph with_weights (my_graph.edges as_map (each::
   geometry(each).perimeter));
```

The calculation of shortest path can change according to the weight you choose for your edges. For example, here is the result of the calculation of the shortest path when all the edges have 1 as weight value (it is the default graph topology), and when the edges have their length as weight.

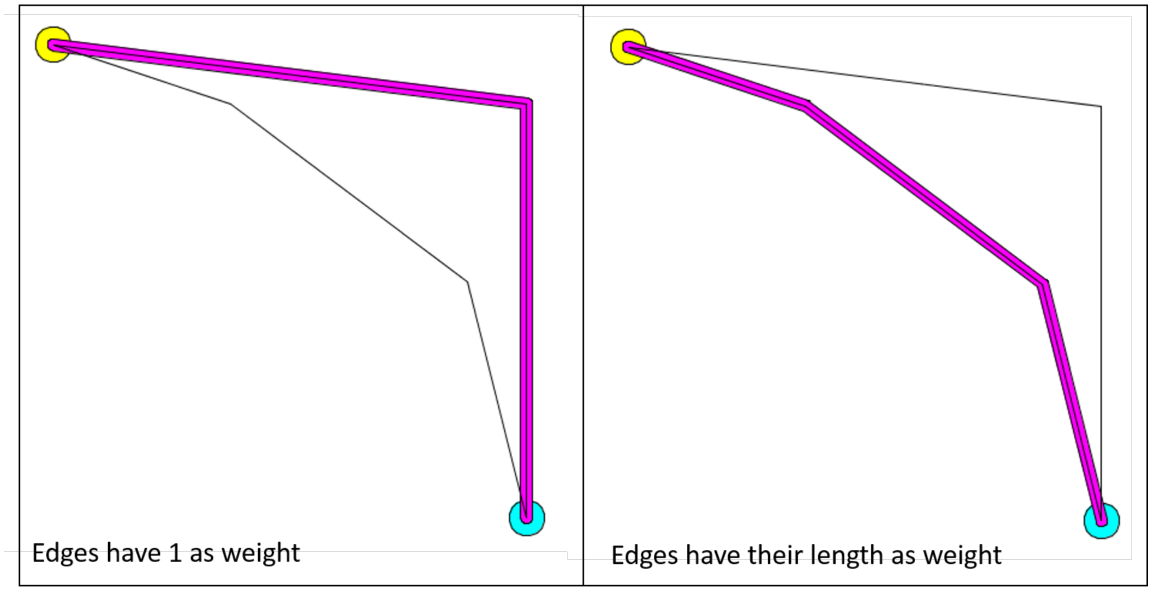


Figure 30.5: path_weight.png

Here is an example of implementation:

```
model shortest_path_with_weight

global
{
    graph my_graph<-spatial_graph([]);
    path shortest_path;
    list<point> nodes;

    init
    {
        add point(10.0,10.0) to:nodes;
        add point(90.0,90.0) to:nodes;
        add point(40.0,20.0) to:nodes;
        add point(80.0,50.0) to:nodes;
        add point(90.0,20.0) to:nodes;

        loop nod over:nodes {
            my_graph <- my_graph add_node(nod);
        }

        my_graph <- my_graph add_edge (nodes at 0::nodes at 2);
        my_graph <- my_graph add_edge (nodes at 2::nodes at 3);
        my_graph <- my_graph add_edge (nodes at 3::nodes at 1);
        my_graph <- my_graph add_edge (nodes at 0::nodes at 4);
        my_graph <- my_graph add_edge (nodes at 4::nodes at 1);

        // comment/decomment the following line to see the
   difference.
        my_graph <- my_graph with_weights (my_graph.edges as_map
   (each::geometry(each).perimeter));

        shortest_path <- path_between(my_graph,nodes at 0, nodes
   at 1);
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            graphics "shortest path" {
                if (shortest_path != nil) {
                    draw circle(3) at: point(shortest_path.source
   ) color: #yellow;
                    draw circle(3) at: point(shortest_path.target
   ) color: #cyan;
                    draw (shortest_path.shape+1) color: #magenta;
                }
                loop edges over: my_graph.edges {
                    draw edges color: #black;
                }
            }
        }
    }
}
```

Chapter 31

Mirror species

A mirror species is a species whose population is automatically managed with respect to another species. Whenever an agent is created or destroyed from the other species, an instance of the mirror species is created or destroyed. Each of these 'mirror agents' has access to its reference agent (called its target). Mirror species can be used in different situations but the one we describe here is more oriented towards visualization purposes.

Index

Declaration

A mirror species can be defined using the `mirrors` keyword as following:

```
species B mirrors: A{
}
```

In this case the species B mirrors the species A.

By default the location of the species B will be random but in many cases, once want to place the mirror agent at the same location of the reference species. This can be achieve by simply adding the following lines in the mirror species :

```
species B mirrors: A{
    point location <- target.location update: target.location;
}
```

`target` is a built-in attribute of a mirror species. It refers to the instance of the species tracked.

In the same spirit any attribute of a reference species can be reach using the same syntax. For instance if the species A has an attribute called `attribute1` of type `int` is is possible to get this attribute from the mirror species B using the following syntax:

```
int value <- target.attribute1;
```

Example

To practice a bit with the mirror notion, we will now build a simple model displaying a species A (aspect: white circle) moving randomly, and another species B (aspect: blue sphere) with the species A location on x and y, with an upper value for the z axis.

Here is an example of implementation for this model:

```
model Mirror

global {
  init{
    create A number:100;
  }
}

species A skills:[moving]{
    reflex update{
        do wander;
    }
    aspect base{
        draw circle(1) color: #white;
    }
}
species B mirrors: A{
```

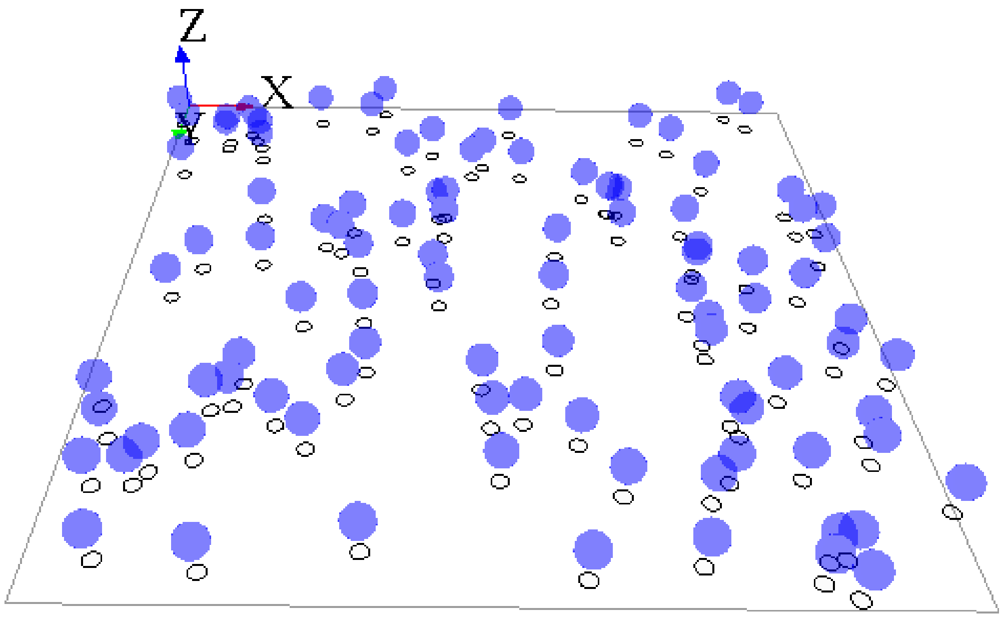


Figure 31.1: images/mirror_model.png

```
    point location <- target.location update: point(target.
   location.x,target.location.y,target.location.z+5);
    aspect base {
        draw sphere(2) color: #blue;
    }
}

experiment mirroExp type: gui {
    output {
        display superposedView type: opengl{
          species A aspect: base;
          species B aspect: base transparency:0.5;
        }
    }
}
```

Chapter 32

Multi-level architecture

The multi-level architecture offers the modeler the following possibilities: the declaration of a species as a micro-species of another species, the representation of an entity as different types of agent (i.e., GAML species), the dynamic migration of agents between populations.

Index

Declaration of micro-species

A species can have other species as micro-species. The micro-species of a species is declared inside the species' declaration.

```
species macro_species {
      species micro_species_in_group {
      }
}
```

In the above example, “micro_species_in_group” is a micro-species of “macro_species”. An agent of “macro_species” can have agents “micro_species_in_group” as micro-agents. Agents of “micro_species_in_group” have an agent of “macro_species” as “host” agent.

As the species “micro_species_in_group” is declared inside the species “macro_species”, “micro_species_in_group” will return a list of “micro_species_in_group” agent inside the given “macro_species” agent.

```
global
{
    init {
        create macro_species number:5;
    }
}

species macro_species
{
    init {
        create micro_species_in_group number:rnd(10);
        write "the macro species named "+name+" contains "+length
    (micro_species_in_group)+" micro-species.";
    }

    species micro_species_in_group {
    }
}

experiment my_experiment type: gui {
}
```

In this above example, we create 5 macro-species, and each one of these macro-species create a random number of inner micro-species. We can see that “micro_species_in_group” refers to the list of micro-species inside the given macro-species.

Access to micro-agents, host agent

To access to micro-agents (from a macro-agent), and to host agent (from a micro-agents), you have to use two built-in attributes.

The `members` built-in attribute is used inside the macro-agent, to get the list of all its micro-agents.

```
species macro_species
{
    init {
        create first_micro_species number:3;
        create second_micro_species number:6;
        write "the macro species named "+name+" contains "+length
   (members)+" micro-species.";
    }

    species first_micro_species {
    }

    species second_micro_species {
    }
}
```

The `host` built-in attribute is used inside the micro-agent to get the host macro-agent.

```
species macro_species {

    micro_species_in_group micro_agent;

    init {
        create micro_species_in_group number:rnd(10);
        write "the macro species named "+name+" contains "+length
   (members)+" micro-species.";
    }

    species micro_species_in_group {
        init {
            write "the micro species named "+name+" is hosted by
   "+host;
        }
    }
}
```

NB: We already said that the world agent is a particular agent, instantiated just once. In fact, the world agent is the host of all the agents. You can try to get the host for a regular species, you will get the world agent itself (named as you named your model). You can also try to get the members of your world (from the global scope for example), and you will get the list of the agents presents in the world.

```
global
{
    init {
        create macro_species number:5;
        write "the world has "+length(members)+" members.";
    }
}

species macro_species
{
    init {
        write "the macro species named "+name+" is hosted by "+
   host;
    }
}
```

Representation of an entity as different types of agent

The multi-level architecture is often used in order to represent an entity through different types of agent. For example, an agent “bee” can have a behavior when it is alone, but when the agent is near from a lot of agents, he can changes his type to “bee_in_swarm”, defined as a micro-species of a macro-species “swarm”. Other example: an agent “pedestrian” can have a certain behavior when walking on the street, and then change his type to “pedestrian_-in_building” when he is in a macro-species “building”. You have then to distinguish two different species to define your micro-species: - The first can be seen as a regular species (it is the “bee” or the “pedestrian” for instance). We will name this species as “micro_species”. - The second is the real micro-species, defined inside the macro-species (it is the “bee_in_-swarm” or the “pedestrian_in_building” for instance). We will name this species as “micro_-species_in_group”. This species has to inherit from the “micro_species”.

```
species micro_species {
}

species macro_species
{
    species micro_species_in_group parent: micro_species {
    }
}
```

Dynamic migration of agents

In our example about bees, a "swarm" entity is composed of nearby flying "bee" entities. When a "bee" entity approaches a "swarm" entity, this "bee" entity will become a member of the group. To represent this, the modeler lets the "bee" agent change its species to "bee_-in_swarm" species. The "bee" agent hence becomes a "bee_in_swarm" agent. To change species of agent, we can use one of the following statements: `capture`, `release`, `migrate`.

The statement `capture` is used by the "macro_species" to capture one (or several) "micro_-species" agent(s), and turn it (them) to a "micro_species_in_group". You can specify which agent (or list of agents) you want to capture by using the facet `target`. The facet `as` is used to cast the agent from "micro_species" to "micro_species_in_group". You can use the facet `return` to get the newly captured agent(s).

```
capture target:micro_species as:micro_species_in_group;
```

The statement `release` is used by the "macro_species" to release one (or several) "micro_-species_in_group" agent(s), and turn it (them) to a "micro_species". You can specify which agent (or list of agents) you want to release by using the facet `target`. The facet `as` is used to cast the agent from "micro_species_in_group" to "micro_species". The facet `in` is used to specify the new host (by default, it is the host of the "macro_species"). You can use the facet `return` to get the newly released agent(s).

```
release target:list(micro_species_in_group) as:micro_species in:
   world;
```

The statement `migrate`, less used, permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Read the GAML Reference to learn more about this statement. //: # (endConcept|multi_level)

Example:

Here is an example of micro_species that gather together in macro_species when they are close enough.

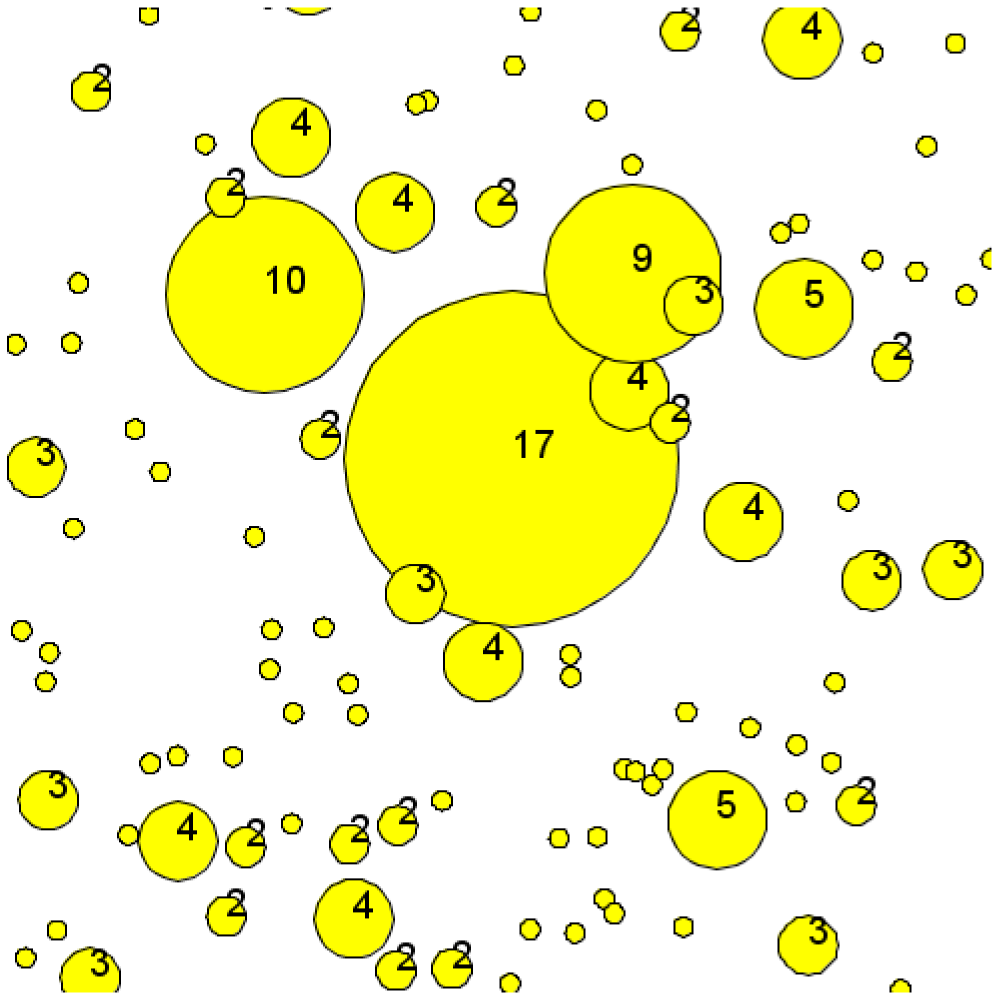


Figure 32.1: images/multilevel_model.png

```
model multilevel

global {
    int release_time <- 20;
    int capture_time <- 100;
    int remaining_release_time <- 0;
    int remaining_capture_time <- capture_time;
    init {
        create micro_species number:200;
    }
    reflex reflex_timer {
        if (remaining_release_time=1)
        {
            remaining_release_time <- 0;
            remaining_capture_time <- capture_time;
        }
        else if (remaining_capture_time=1)
        {
            remaining_capture_time <- 0;
            remaining_release_time <- release_time;
        }
        remaining_release_time <- remaining_release_time - 1;
        remaining_capture_time <- remaining_capture_time - 1;
    }
    reflex capture_micro_species when:(remaining_capture_time>0
   and flip(0.1)) {
        ask macro_species {
            list<micro_species> micro_species_in_range <-
   micro_species at_distance 1;
            if (micro_species_in_range != []) {
                do capture_micro_species(micro_species_in_range);
            }
        }
        ask micro_species {
            list<micro_species> micro_species_list_to_be_captured
    <- micro_species at_distance 1;
            if(micro_species_list_to_be_captured != []) {
                create macro_species {
                    location <- myself.location;
                    add item:myself to:
```

```
   micro_species_list_to_be_captured;
                    do capture_micro_species(
   micro_species_list_to_be_captured);
                }
            }
        }
    }
}

species micro_species skills:[moving] {
    geometry shape <- circle(1);
    aspect base {
        draw shape;
    }
    reflex move{
        do wander;
    }
}

species macro_species {
    geometry shape <- circle(1) update:circle(length(members));

    species micro_species_in_group parent:micro_species {
    }

    action capture_micro_species(list<micro_species> micro_list)
   {
        loop mic_sp over:micro_list {
            capture mic_sp as:micro_species_in_group;
        }
    }

    reflex release_reflex when:(remaining_release_time>0 and flip
   (0.1)) {
        release members as:micro_species /*in:world*/;
        do die;
    }

    aspect base {
        draw shape;
        draw text:string(length(members)) color:#black size:4;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species macro_species aspect: base;
            species micro_species aspect: base;
        }
    }
}
```

Defining GUI Experiments

When you execute your simulation, you will often need to display some information. For each simulation, you can define some inputs and outputs: * The inputs will be composed of parameters manipulated by the user for each simulation. * The outputs will be composed of displays, monitors or output files. They will be define inside the scope `output`.

```
experiment exp_name type: gui {
   [input]
   output {
     [display statements]
     [monitor statements]
     [file statements]
   }
}
```

You can define two types of experiment (through the facet `type`): * `gui` experiments (the default type) are used to play an experiment, and interpret its outputs. * `batch` experiments are used to play an experiment several times (usually with other input values), used for model exploration. We will come back to this notion a bit further in the tutorial.

Inside experiment scope, you can access to some built-ins which can be useful, such as `minimum_cycle_duration`, to force the duration of one cycle.

```
experiment my_experiment type: gui {
    float minimum_cycle_duration <- 2.0#minute;
}
```

Other built-ins are available, to learn more about, go to the page **experiment built-in**.

In this part, we will focus on the **gui experiments**. We will start with learning how to **define input parameters**, then we will study the outputs, such as **displays**, **monitors and inspectors**, and **export files**. We will finish this part with how to define **user commands**. //: # (endConcept|gui_experiments)

Chapter 33

Defining Parameters

When playing simulation, you have the possibility to define input parameters, in order to change them and replay the simulation. Defining parameters allows to make the value of a global variable definable by the user through the user graphic interface.

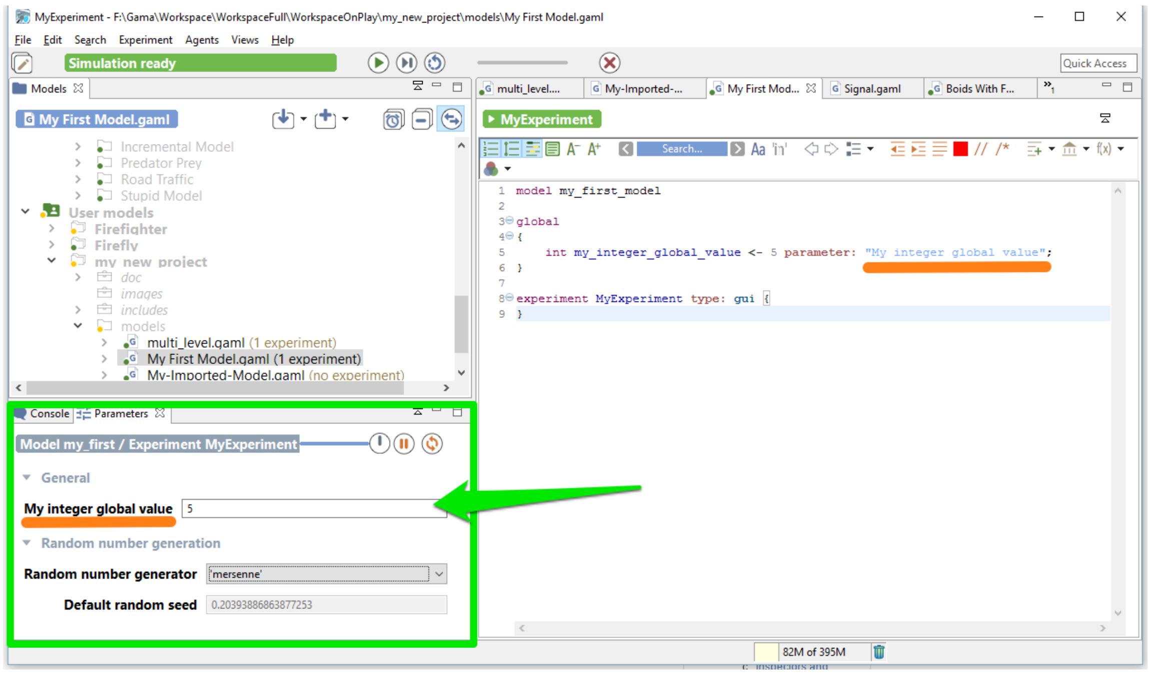
Index

Defining parameters

You can define parameters inside the global scope, when defining your global variables with the facet `parameter`:

```
global
{
    int my_integer_global_value <- 5 parameter: "My integer
   global value";
}
```

When launching your experiment, the parameter will appear in your "Parameters" panel, with the name you chose for the `parameter` facet.



```
model my_first_model

global
{
    int my_integer_global_value <- 5 parameter: "My integer global value";
}

experiment MyExperiment type: gui {
}
```

Model my_first / Experiment MyExperiment

General

My integer global value: 5

Random number generation

Random number generator: 'mersenne'

Default random seed: 0.20393886863877253

Figure 33.1: images/parameter1.png

You can also define your parameter inside the experiment, using the statement parameter. You have to specify first the name of your parameter, then the name of the global variable through the facet `var`.

```
global
{
    int my_integer_global_value <- 5;
}
```

experiment MyExperiment type: gui { parameter "My integer global value" var:my_integer_global_value; }

NB: This variable has to be initialized with a value. If you don't want to initialize your value on the `global` scope, you can initialize the value directly on the parameter statement, using the facet `init`.

```
global
{
    int my_integer_global_value;
}
```

```
experiment MyExperiment type: gui {
    parameter "My integer global value" var:
   my_integer_global_value init:5;
}
```

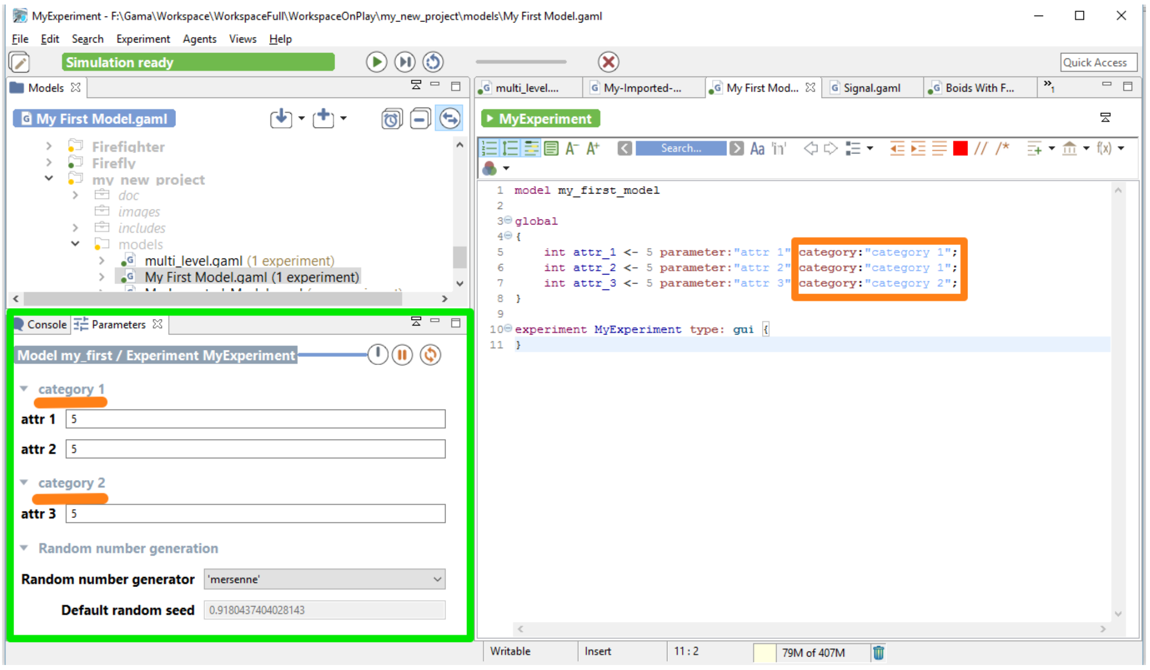
Additional facets

You can use some facets to arrange your parameters. For example, you can categorize your parameters under a label, using the facet `category`:

```
global
{
    int attr_1 <- 5 parameter:"attr 1" category:"category 1";
    int attr_2 <- 5 parameter:"attr 2" category:"category 1";
    int attr_3 <- 5 parameter:"attr 3" category:"category 2";
}
```

You also can add some facets such as `min`, `max` or `among` to improve the declaration of the parameter.

```
global
{
    string fruit <- "none" among:["none","apple","banana"]
   parameter:"fruit" category:"food";
    string vegetable <- "none" among:["none","cabbage","carrot"]
   parameter:"vegetable" category:"food";
    int integer_variable <- 5 parameter:"integer variable" min:0
   max:100  category:"other";
}

experiment MyExperiment type: gui {
}
```



```
model my_first_model

global
{
    int attr_1 <- 5 parameter:"attr 1" category:"category 1";
    int attr_2 <- 5 parameter:"attr 2" category:"category 1";
    int attr_3 <- 5 parameter:"attr 3" category:"category 2";
}

experiment MyExperiment type: gui {
}
```

Model my_first / Experiment MyExperiment

category 1

attr 1	5
attr 2	5

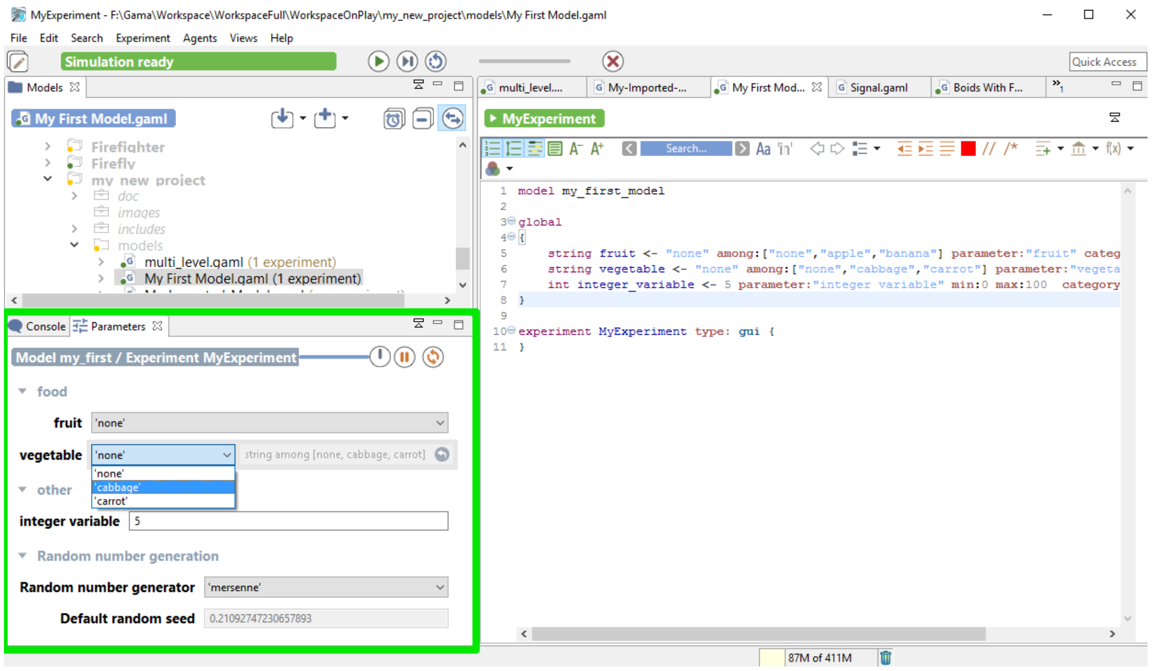
category 2

attr 3	5

Random number generation

Random number generator	'mersenne'
Default random seed	0.9180437404028143

Figure 33.2: images/parameter2.png



```
model my_first_model

global
{
    string fruit <- "none" among:["none","apple","banana"] parameter:"fruit" categ
    string vegetable <- "none" among:["none","cabbage","carrot"] parameter:"vegeta
    int integer_variable <- 5 parameter:"integer variable" min:0 max:100  category
}

experiment MyExperiment type: gui {
}
```

Model my_first / Experiment MyExperiment

food

fruit	'none'
vegetable	'none'

string among [none, cabbage, carrot]

'none'
'cabbage'
'carrot'

other

integer variable	5

Random number generation

Random number generator	'mersenne'
Default random seed	0.21092747230657893

//: # (endConcept|define_parameters)

Chapter 34

Defining displays (Generalities)

Index

Displays and layers

A display is the graphical output of your simulation. You can define several displays related with what you want to represent from your model execution. To define a display, use the keyword `display` inside the `output` scope, and specify a name (`name` facet).

```
experiment my_experiment type: gui {
    output {
        display "display1" {
        }
        display name:"display2" {
        }
    }
}
```

Other facets are available when defining your display: * Use `background` to define a color for your background

```
display "my_display" background:#red
```

- Use `refresh` if you want to refresh the display when a condition is true (to refresh your display every number of steps, use the operator `every`)

  ```
  display "my_display" refresh:every(10)
  ```

You can choose between two types of displays, by using the facet type: * java2D displays will be used when you want to have 2D visualization. It is used for example when you manipulate charts. This is the default value for the facet type. * opengl displays allows you to have 3D visualization.

You can save the display on the disk, as a png file, in the folder name_of_model/models/s-napshots, by using the facet `autosave`. This facet takes one a boolean as argument (to allow or not to save each frame) or a point (to define the size of your image). By default, the resolution of the output image is 500x500px (note that when no unit is provided, the unit is `#px` (pixel)).

```
display my_display autosave:true type:java2D {}
```

is equivalent to :

```
display my_display autosave:{500,500} type:java2D {}
```

Each display can be decomposed in one or several layers. Here is a screenshot (from the Toy Model Ant) to better understand those different notions we are about to tackle in this session.

Organize your layers

In one 2D display, you will have several types of layers, giving what you want to display in your model. You have a large number of layers available. You already know some of them,

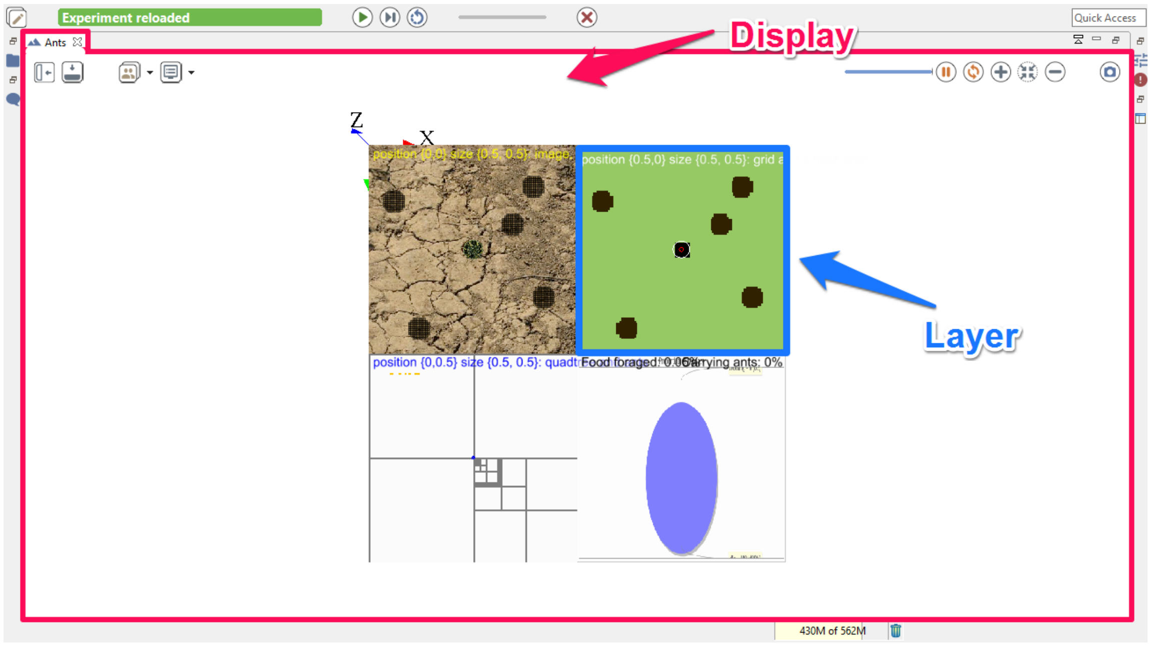

Experiment reloaded
Quick Access
Ants
Display
Layer
Z
X
position {0,0} size {0.5, 0.5}: image
position {0.5,0} size {0.5, 0.5}: grid
position {0,0.5} size {0.5, 0.5}: quadt
Food foraged: 0.00% Carrying ants: 0%
430M of 562M

Figure 34.1: images/difference_layer_display.png

such as `species`, `agents`, `grid`, but other specific layers such as `image` (to display image) and `graphics` (to freely draw shapes/geometries/texts without having to define a species) are also available

Each layer will be displayed in the same order as you declare them. The last declared layer will be above the others.

Thus, the following code:

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" {
                draw square(20) at:{10,10} color:#gold;
            }
            graphics "layer2" {
                draw square(20) at:{15,15} color:#darkorange;
            }
            graphics "layer3" {
                draw square(20) at:{20,20} color:#cornflowerblue;
            }
        }
    }
}
```

Will have this output:

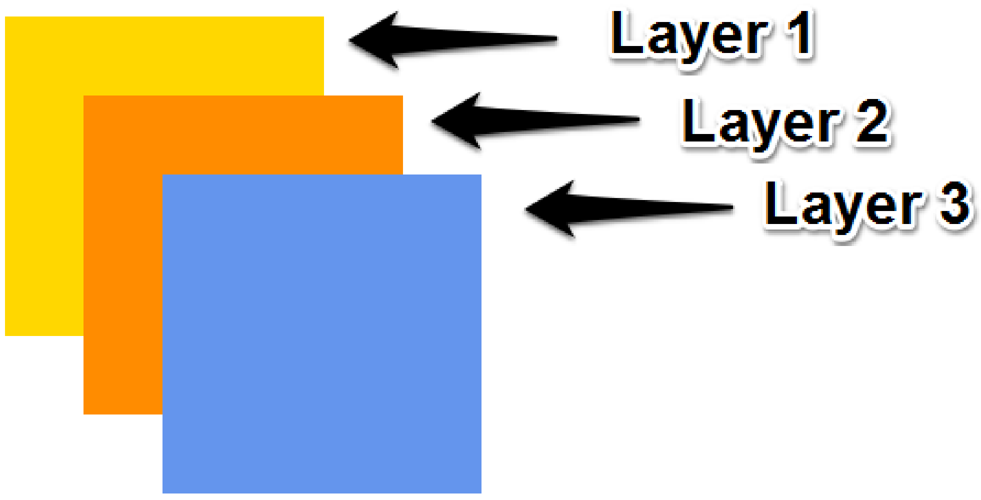


Figure 34.2: images/layers_order.png

Most of the layers have the `transparency` facet in order to see the layers which are under.

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" {
                draw square(20) at:{10,10} color:#darkorange;
            }
            graphics "layer2" transparency:0.5 {
                draw square(20) at:{15,15} color:#cornflowerblue;
            }
        }
    }
}
```

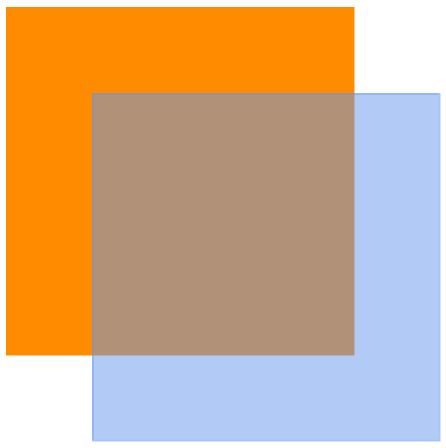


Figure 34.3: images/layers_transparency.png

To specify a position and a size for your layer, you can use the `position` and the `size` facets. The `position` facet is used with a point type, between {0,0} and {1,1}, which corresponds to the position of the upper left corner of your layer in percentage. Then, if you choose the point {0.5,0.5}, the upper left corner of your layer will be in the center of your display. By default, this value is {0,0}. The `size` facet is used with a point type, between {0,0} and {1,1} also. It corresponds to the size occupied by the layer in percentage. By default, this value is {1,1}.

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" position:{0,0} size:{0.5,0.8} {
                draw shape color:#darkorange;
            }
            graphics "layer2" position:{0.3,0.1} size:{0.6,0.2} {
                draw shape color:#cornflowerblue;
            }
            graphics "layer3" position:{0.4,0.2} size:{0.3,0.8} {
                draw shape color:#gold;
            }
        }
    }
}
```

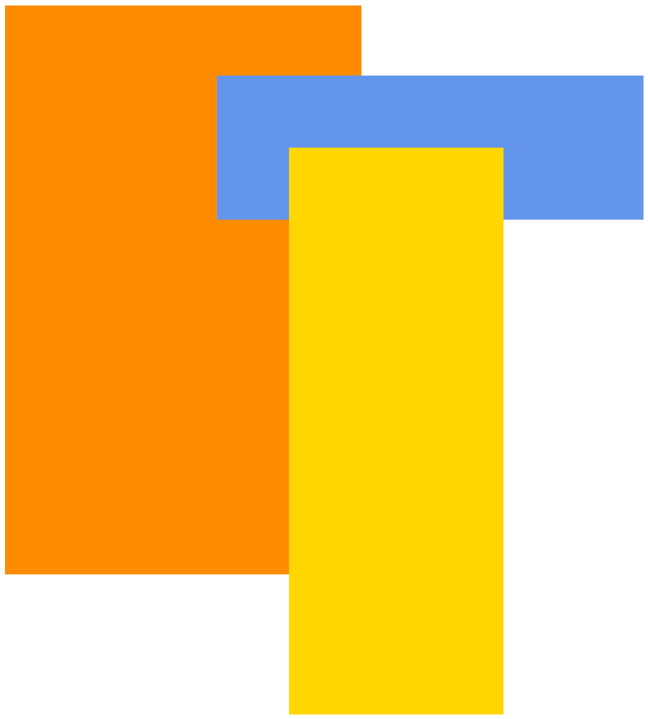


Figure 34.4: images/layers_size_position.png

NB : `displays` can have background, while `graphics` can't. If you want to put a background for your `graphics`, a solution can be to draw the `shape` of the world (which is, by default, a square 100m*100m).

A lot of other facets are available for the different layers. Please read the documentation of `graphics` for more information.

Example of layers

agents layer

`agents` allows the modeler to display only the agents that fulfill a given condition.

Please read the documentation about `agents` statement if you are interested.

species layer

`species` allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

Please read the documentation about `species` statement if you are interested.

image layer

`image` allows modeler to display an image (e.g. as background of a simulation).

Please read the documentation about `image` statement if you are interested.

graphics layer

`graphics` allows the modeler to freely draw shapes/geometries/texts without having to define a species.

Please read the documentation about `graphics` statement if you are interested.

Chapter 35

Defining Charts

To visualize result and make analysis about you model, you will certainly have to use charts. You can define 3 types of charts in GAML: histograms, pie, and series. For each type, you will have to determine the data you want to highlight.

Index

Define a chart

To define a chart, we have to use the `chart` statement. A chart has to be named (with the `name` facet), and the type has to be specified (with the `type` facet). The value of the `type` facet can be `histogram`, `pie`, `series`, `scatter`, `xy`. A chart has to be defined inside a display.

```
experiment my_experiment type: gui {
    output {
        display "my_display" {
            chart "my_chart" type:pie {
            }
        }
    }
}
```

After declaring your chart, you have to define the data you want to display in your chart.

Data definition

Data can be specified with: * several data statements to specify each series * one datalist statement to give a list of series. It can be useful if the number of series is unknown, variable or too high.

The `data` statement is used to specify which variable will be displayed. You have to give your data a name (that will be displayed in your chart), the value of the variable you want to follow (using the `value` facet). You can add come optional facets such as `color` to specify the color of your data.

```
global
{
    int numberA <- 2 update:numberA*2;
    int numberB <- 10000 update:numberB-1000;
}

experiment my_experiment type: gui {
    output {
        display "my_display" {
            chart "my_chart" type:pie {
                data "numberA" value:numberA color:#red;
                data "numberB" value:numberB color:#blue;
            }
        }
    }
}
```

(TODO_IMAGE)

The `datalist` statement is used several variables in one statement. Instead of giving simple values, datalist is used with lists.

```
datalist ["numberA","numberB"] value:[numberA,numberB] color:[#
   red,#blue];
```

[TODO] Datalist provides you some additional facets you can use. If you want to learn more about them, please read the documentation [URL]

Different types of chart

As we already said, you can display 3 types of graphs: the histograms, the pies and the series.

The histograms

[TODO]

Chapter 36

Defining 3D Displays

Table of contents

OpenGL display

- Define the attribute type of the display with `type:opengl` in the output of your model (or use the preferences->display windows to use it by default):

```
output {
  display DisplayName type:opengl {
species mySpecies;
  }
```

The opengl display share most of the feature that the java2D offers and that are described here.

Using 3D display offers many way to represent a simulation. A layer can be positioned and scaled in a 3D world. It is possible to superpose layer on different z value and display different information on the model at different position on the screen.

Position

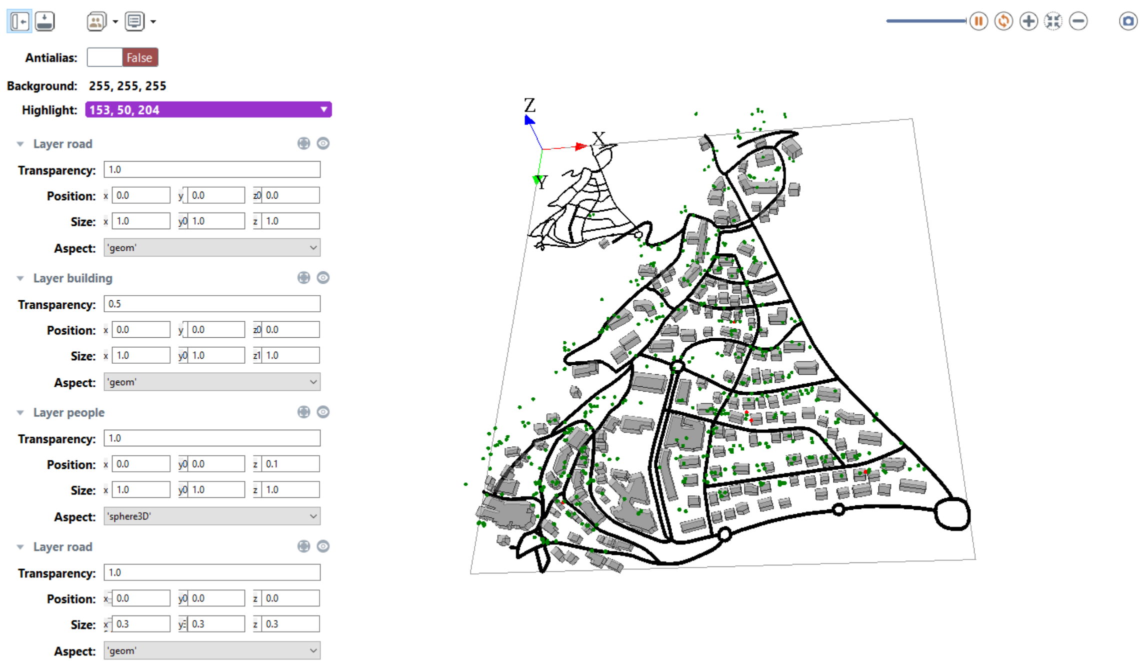
Layer can be drawn on different position (x,y and z) value using the *position* facet

Size

Layer can be drawn with different size (x,y and z) using the *size* facet

Here is an example of display using all the previous facet (experiment factice to add to the model *Incremental Model 5*). You can also dynamically change those value by showing the side bar in the display.

```
experiment expe_test type:gui {
    output {
        display city_display type: opengl{
            species road aspect: geom refresh:false;
            species building aspect: geom transparency:0.5 ;
            species people aspect: sphere3D position:{0,0,0.1};
            species road aspect: geom size:{0.3,0.3,0.3};
        }
    }
}
```


Antialias: False

Background: 255, 255, 255

Highlight: 153, 50, 204

Layer road

Transparency: 1.0

Position: x 0.0 y 0.0 z0 0.0

Size: x 1.0 y0 1.0 z 1.0

Aspect: 'geom'

Layer building

Transparency: 0.5

Position: x 0.0 y 0.0 z0 0.0

Size: x 1.0 y0 1.0 z1 1.0

Aspect: 'geom'

Layer people

Transparency: 1.0

Position: x 0.0 y0 0.0 z 0.1

Size: x 1.0 y0 1.0 z 1.0

Aspect: 'sphere3D'

Layer road

Transparency: 1.0

Position: x 0.0 y0 0.0 z 0.0

Size: x 0.3 y 0.3 z 0.3

Aspect: 'geom'

Figure 36.1: images/species_layer.png

Camera

<a href='http://www.youtube.com/watch?feature=player_embedded&v=rMIVQlul1Ag' target='_blank'>

Arcball Camera

FreeFly Camera

Dynamic camera

User have the possibility to set dynamically the parameter of the camera (observer). The basic camera properties are its **position**, the **direction** in which is pointing, and its **orientation**. Those 3 parameters can be set dynamically at each iteration of the simulation.

Camera position

The facet `camera_pos(x,y,z)` places the camera at the given position. The default camera positon is *(world.width/2,world/height/2,world.maxDim*1.5)* to place the camera at the middle of the environement at an altitude that enables to see the entire environment.

Camera direction (Look Position)

The facet `camera_look_pos(x,y,z)` points the camera toward the given position. The default look position is *(world.width/2,world/height/2,0)* to look at the center of the environment.

Camera orientation (Up Vector)

The camera `camera_up_vector(x,y,z)` sets the *up vector* of the camera. The *up vector* direction in your scene is the *up* direction on your display screen. The default value is (0,1,0)

Here are some examples that can be done using those 3 parameters. You can test it by running the following model:

<a href='http://www.youtube.com/watch?feature=player_embedded&v=lQVGD8aDKZY' target='_blank'>

Boids 3D Camera movement

Default view

```
display RealBoids   type:opengl{
...
}
```

First person view

You can set the position as a first person shooter video game using:

```
display FirstPerson  type:opengl
camera_pos:{boids(1).location.x,-boids(1).location.y,10}
camera_look_pos:{cos(boids(1).heading)*world.shape.width,-sin(
   boids(1).heading)*world.shape.height,0}
camera_up_vector:{0.0,0.0,1.0}{
...
}
```

Third Person view

You can follow an agent during a simulation by positioning the camera above it using:

```
display ThirdPerson  type:opengl camera_pos:{boids(1).location.x
   ,-boids(1).location.y,250}  camera_look_pos:{boids(1).location
   .x,-boids(1).location.y,boids(1).location.z}{
...
}
```

Lighting

In a 3D scene once can define light sources. The way how light sources and 3D object interact is called lighting. Lighting is an important factor to render realistic scenes.

In a real world, the color that we see depend on the interaction between color material surfaces, the light sources and the position of the viewer. There are four kinds of lighting called *ambient*, *diffuse*, *specular* and *emissive*.

Gama handle *ambient* and *diffuse* light.

- **ambient_light**: Allows to define the value of the ambient light either using an int (ambient_light:(125)) or a rgb color ((ambient_light:rgb(255,255,255)). default is rgb(125,125,125).
- **diffuse_light**: Allows to define the value of the diffuse light either using an int (diffuse_light:(125)) or a rgb color ((diffuse_light:rgb(255,255,255)). default is rgb(125,125,125).
- **diffuse_light_pos**: Allows to define the position of the diffuse light either using an point (diffuse_light_pos:{x,y,z}). default is {world.shape.width/2,world.shape.height/2,world.shape.width*2}.
- **is_light_on**: Allows to enable/disable the light. Default is true.
- **draw_diffuse_light**: Allows to enable/disable the drawing of the diffuse light. Default is false“)),

Here is an example using all the available facet to define a diffuse light that rotate around the world.

<a href=‘http://www.youtube.com/watch?feature=player_embedded&v=op56elmEEYs’ target=’_blank’>

```
display View1  type:opengl draw_diffuse_light:true ambient_light
   :(0) diffuse_light:(255) diffuse_light_pos:{50+ 150*sin(time
   *2),50,150*cos(time*2){
...
}
```

Chapter 37

Defining monitors and inspectors

Other outputs can be very useful to study better the behavior of your agents.

Index

Define a monitor

A monitor allows to follow the value of an arbitrary expression in GAML. It will appear, in the User Interface, in a small window on its own and be recomputed every time step (or according to its refresh facet).

Definition of a monitor:

```
monitor monitor_name value: an_expression refresh:
   boolean_statement;
```

with: * `value`: mandatory, the expression whose value will be displayed by the monitor. * `refresh`: bool statement, optional : the new value is computed if the bool statement returns true.

Example:

```
experiment my_experiment type: gui {
    output {
        monitor monitor_name value: cycle refresh:every(1);
    }
}
```

NB : you can also declare monitors during the simulation, by clicking on the button "Add new monitor", and specifying the name of the variable you want to follow.

Define an inspector

During the simulation, the user interface of GAMA provides the user the possibility to inspect an agent, or a group of agents. But you can also define the inspector you want directly from your model, as an output of the experiment.

Use the statement `inspect` to define your inspector, in the output scope of your gui experiment. The inspector has to be named (using the facet `name`), a value has to be specified (with the `value` facet).

```
inspect name:"inspector_name" value:the_value_you_want_to_display
   ;
```

Note that you can inspect any type of species (regular species, grid species, even the world...).

The optional facet `type` is used to specify the type of your inspector. 2 values are possible : * *agent* (default value) if you want to display the information as a regular agent inspector. Note that if you want to inspect a large number of agents, this can take a lot of time. In this case, prefer the other type *table* * *table* if you want to display the information as an agent browser

The optional facet `attribute` is used to filter the attributes you want to be displayed in your inspector.

Beware : only one agent inspector (`type:agent`) can be used for an experiment. Beside, you can add as many agent browser (`type:table`) as you want for your experiment.

Example of implementation :

```
model new

global {
    init {
        create my_species number:3;
    }
}

species my_species {
    int int_attr <- 6;
    string str_attr <- "my_value";
    string str_attr_not_important <- "blabla";
}

grid my_grid_species width: 10 height: 10 {
    int rnd_value <- rnd(5);
}

experiment my_experiment type:gui {
    output {
        inspect name:"my_species_inspector" value:my_species
   attributes:["int_attr","str_attr"];
        inspect name:"my_species_browser" value:my_species type:
   table;
        inspect name:"my_grid_species_browser" value:5 among
   my_grid_species type:table;
    }
}
```

Another statement, `browse`, is doing a similar thing, but preferring the *table* type (if you want to browse an agent species, the default type will be the *table* type). //: # (endConcept|monitors_and_inspectors)

Chapter 38

Defining export files

Index

The Save Statement

Allows to save data in a file. The type of file can be "shp", "text" or "csv". The `save` statement can be use in an init block, a reflex, an action or in a user command. Do not use it in experiments.

Facets

- `to` (string): an expression that evaluates to an string, the path to the file
- `data` (any type), (omissible) : any expression, that will be saved in the file
- `crs` (any type): the name of the projectsion, e.g. crs:"EPSG:4326" or its EPSG id, e.g. crs:4326. Here a list of the CRS codes (and EPSG id): http://spatialreference.org
- `rewrite` (boolean): an expression that evaluates to a boolean, specifying whether the save will ecrase the file or append data at the end of it
- `type` (an identifier): an expression that evaluates to an string, the type of the output file (it can be only "shp", "text" or "csv")

- `with` (map):

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->"  + name + ":" + location) to: "
   save_data.txt" type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp" with:
   [name::"nameAgent", location::"locationAgent"] crs: "EPSG:4326
   ";
```

Export files in experiment

Displays are not the only output you can manage in GAMA. Saving data to a file during an experiment can also be achieved in several ways, depending on the needs of the modeler. One way is provided by the `save` statement, which can be used everywhere in a model or a species. The other way, described here, is to include an **`output_file`** statement in the output section.

```
output_file name:"file_name" type:file_type data:data_to_write;
```

with:

`file_type`: text, csv or xml `file_name`: string `data_to_write`: string

Example:

```
file name: "results" type: text data: time + "; " + nb_preys + ";
    " + nb_predators refresh:every(2);
```

Each time step (or according to the frequency defined in the `refresh` facet of the file output), a new line will be added at the end of the file. If `rewrite: false` is defined in its facets, a new file will be created for each simulation (identified by a timestamp in its name).

Optionally, a `footer` and a `header` can also be described with the corresponding facets (of type string).

Autosave

Image files can be exported also through the `autosave` facet of the display, as explained in this previous part. //: # (endConcept|export_files)

Chapter 39

Defining user interaction

During the simulation, GAML provides you the possibility to define some function the user can execute during the execution. In this chapter, we will see how to define buttons to execute action during the simulation, how to catch click event, and how to use the user control architecture.

Index

Catch Mouse Event

You can catch mouse event during the simulation using the statement `event`. This statement has 2 required facets: * `name` (identifier) : Specify which event do you want to trigger (among the following values : `mouse_down`, `mouse_down`, `mouse_move`, `mouse_enter`, `mouse_exit`). * `action` (identifier) : Specify the name of the global action to call.

```
event mouse_down action: my_action;
```

The event statement has to be defined in the experiment/output/display scope. Once the event is triggered, the global action linked will be called. The action linked has to have 2 arguments : the location of the click (type point) and the list of agents which are displayed at this position.

```
global
{
    action my_action (point loc, list<my_species> selected_agents
   )
    {
        write "do action";
    }

}

species my_species
{
}

experiment my_experiment type: gui
{
    output
    {
        display my_display
        {
            species my_species;
            event mouse_down action: my_action;
        }
    }
}
```

Define User command

Anywhere in the global block, in a species or in an (GUI) experiment, user_command statements can be implemented. They can either call directly an existing action (with or without arguments) or be followed by a block that describes what to do when this command is run.

Their syntax can be (depending of the modeler needs) either:

```
user_command cmd_name action: action_without_arg_name;
//or
user_command cmd_name action: action_name with: [arg1::val1, arg2
   ::val2];
//or
user_command cmd_name {
   // statements
}
```

For instance:

```
user_command kill_myself action: die;
//or
user_command kill_myself action: some_action with: [arg1::5, arg2
   ::3];
//or
user_command kill_myself {
    do die;
}
```

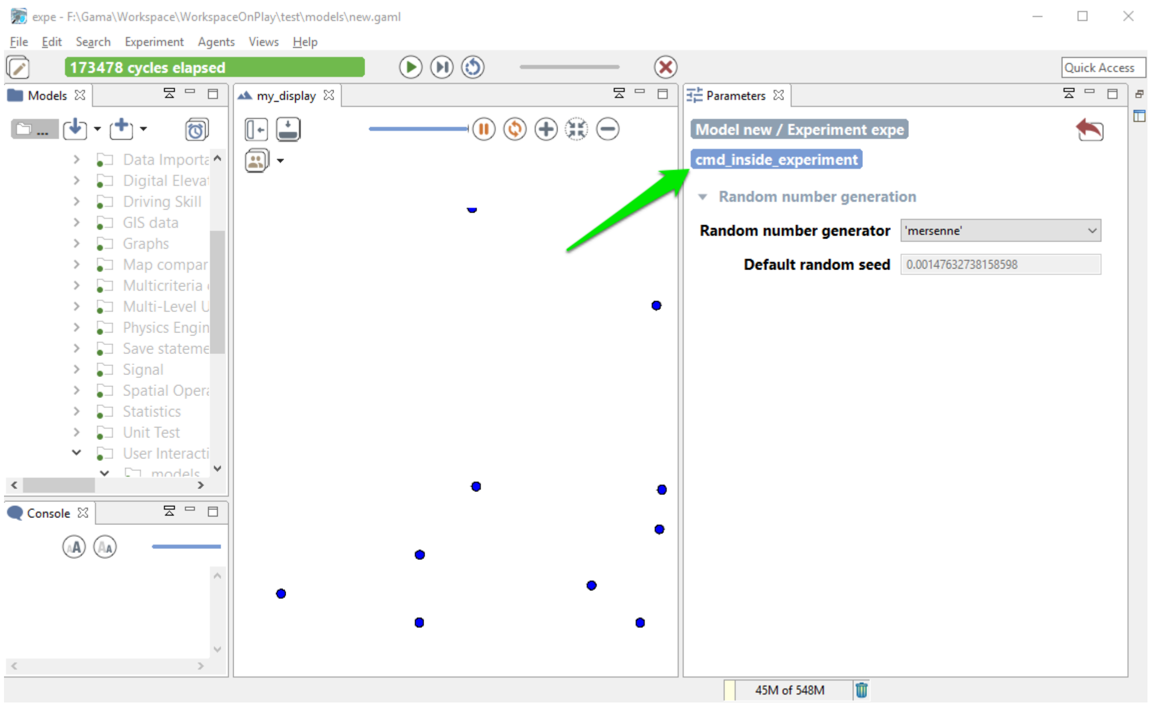
Defining User command in GUI Experiment scope

The user command can be defined directly inside the GUI experiment scope. In that case, the implemented action appears as a button in the top of the parameter view.

Here is a very short code example :

```
model quick_user_command_model

global {
    action createAgent
    {
        create my_species;
    }
}

species my_species {
    aspect base {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    user_command cmd_inside_experiment action:createAgent;
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

And here is screenshots of the execution :



Random number generator 'mersenne'

Default random seed 0.00147632738158598

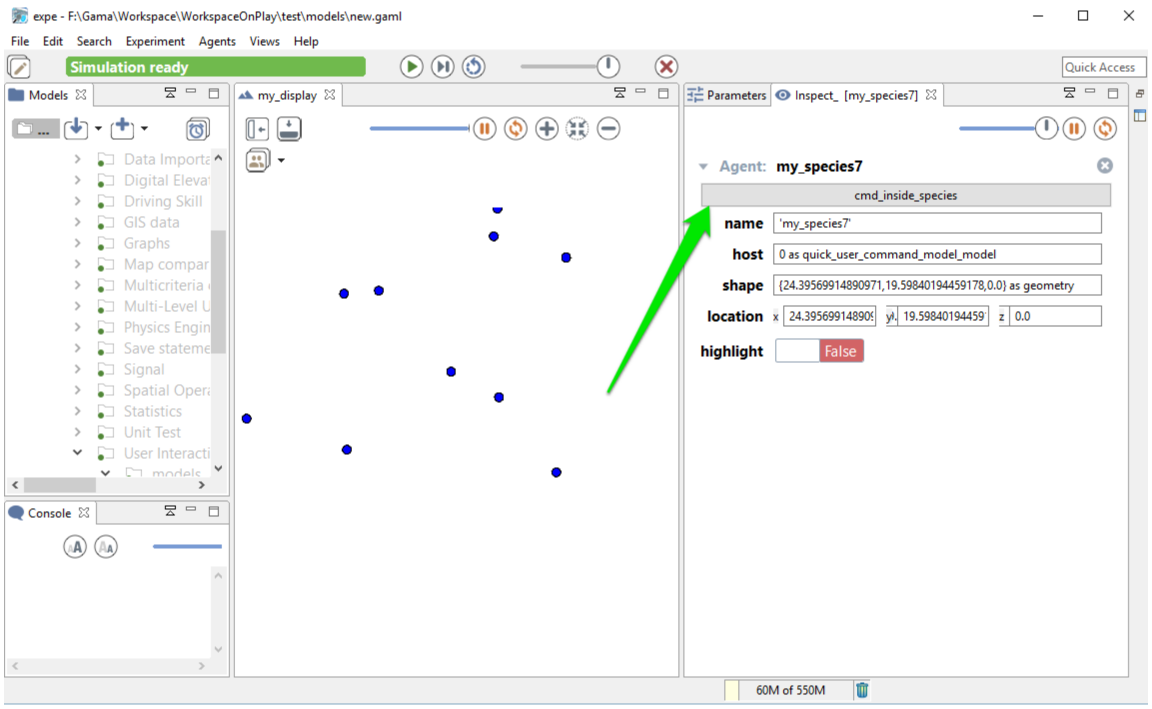
Figure 39.1: images/user_command_inside_expe.png

Defining User command in a global or regular species

The user command can also be defined inside a species scope (either global or regular one). Here is a quick example of model :

```
model quick_user_command_model

global {
    init {
        create my_species number:10;
    }
}

species my_species {
    user_command cmd_inside_experiment action:die;
    aspect base {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

During the execution, you have 2 ways to access to the action : * When the agent is inspected, they appear as buttons above the agents' attributes



Agent: my_species7

cmd_inside_species

name	'my_species7'
host	0 as quick_user_command_model_model
shape	{24.39569914890971,19.59840194459178,0.0} as geometry
location	x 24.395699148909 y 19.59840194459 z 0.0
highlight	False

* When the agent is selected by a right-click in a display, these command appear under the usual "Inspect", "Focus" and "Highlight" commands in the pop-up menu.

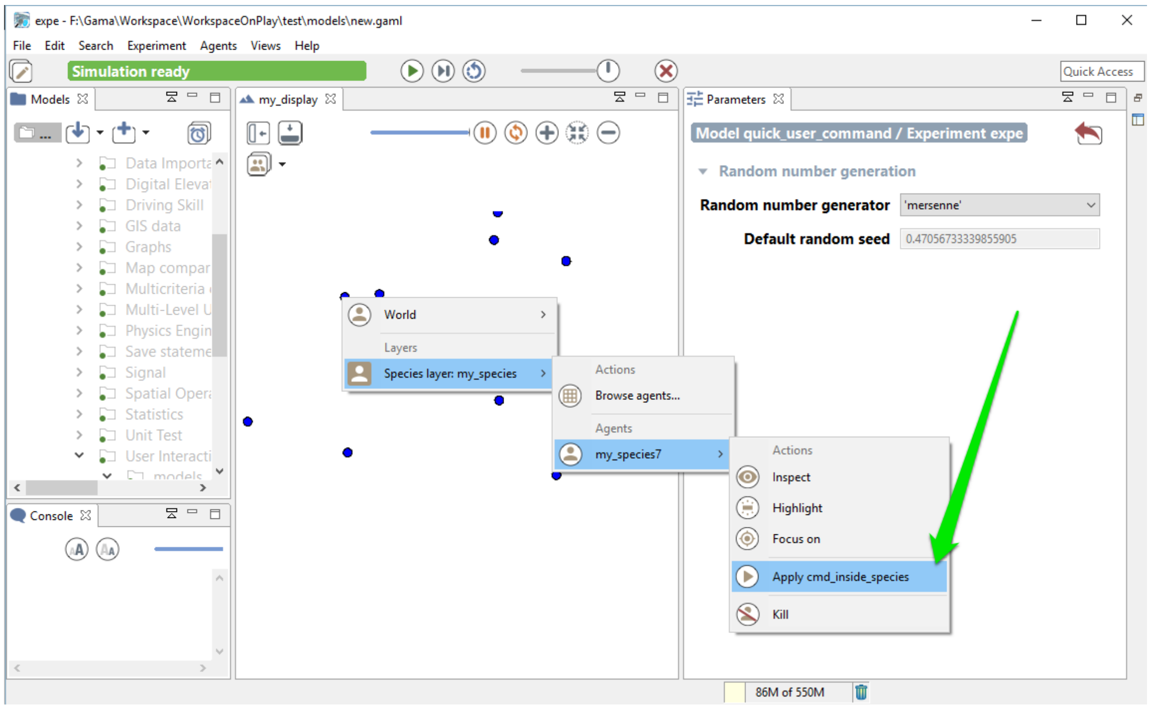
Remark: The execution of a command obeys the following rules: * when the command is called from right-click pop-menu, it is executed immediately * when the command is called from panels, its execution is postponed until the end of the current step and then executed at that time.

user_location

In the special case when the user_command is called from the pop-up menu (from a right-click on an agent in a display), the location chosen by the user (translated into the model coordinates) is passed to the execution scope under the name user_location.

Example:

```
global {
   user_command "Create agents here" {
      create my_species number: 10 with: [location::user_location
   ];
   }
```



Random number generator: 'mersenne'

Default random seed: 0.47056733339855905

Figure 39.2: images/user_command_inside_species2.png

```
}
```

This will allow the user to click on a display, choose the world (always present now), and select the menu item "Create agents here".

Note that if the world is inspected (this `user_command` appears thus as a button) and the user chooses to push the button, the agent will be created at a random location.

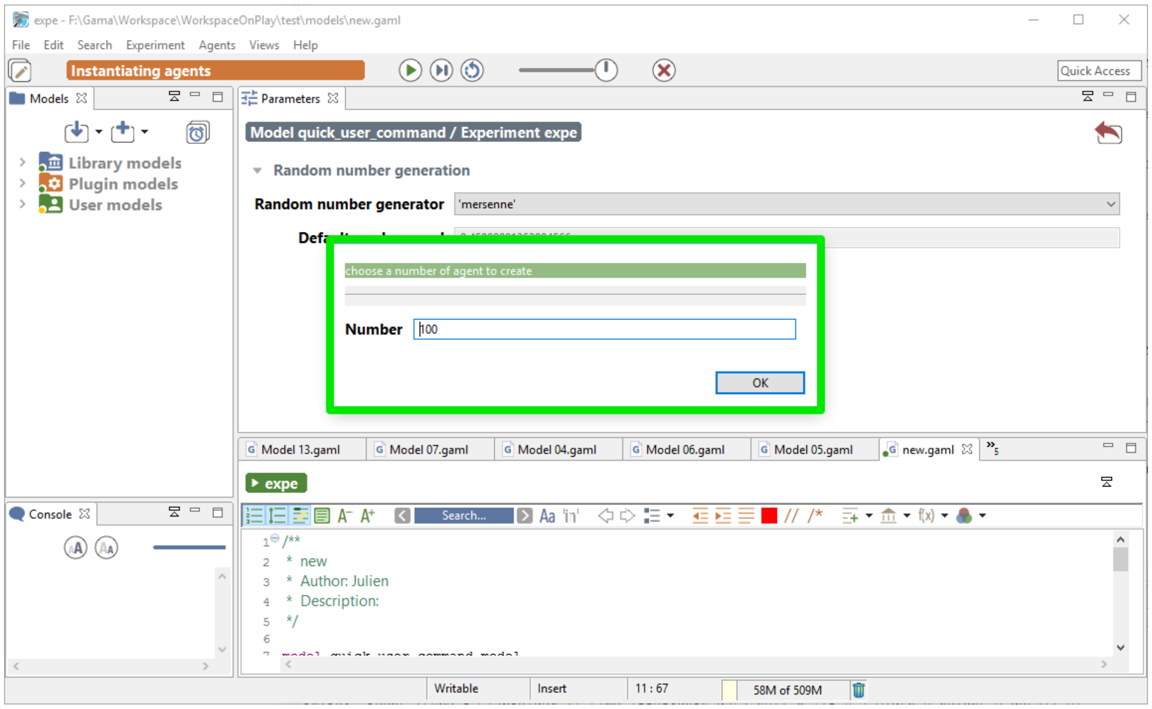
user_input

As it is also, sometimes, necessary to ask the user for some values (not defined as parameters), the `user_input` unary operator has been introduced. This operator takes a map [string::value] as argument (the key is the name of the chosen parameter, the value is the default value), displays a dialog asking the user for these values, and returns the same map with the modified values (if any). You can also add a text as first argument of the operator, which will be displayed as a title for your dialog popup. The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section like the following one to force the user to input new values instead of relying on the initial values of parameters.

Here is an example of implementation:

```
model quick_user_command_model

global {
   init {
      map values <- user_input("choose a number of agent to
   create",["Number" :: 100]);
      create my_species number : int(values at "Number");
   }
}

species my_species {
    aspect base {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

When running this model, you will first have to input a number:



Model quick_user_command / Experiment expe

Random number generation

Random number generator 'mersenne'

choose a number of agent to create

Number 100

OK

```
/**
 * new
 * Author: Julien
 * Description:
 */

model quick_user_command_model
```

Writable Insert 11 : 67 58M of 509M

Figure 39.3: images/input_cmd.png

User Control Architecture

An other way to define user interaction is to use the user control architecture. Please jump directly to the section user control architecture if you want to learn more about this point.

Exploring Models

We just learnt how to launch GUI Experiments from GAMA. A GUI Experiment will start with a particular set of input, compute several outputs, and will stop at the end (if asked).

In order to explore models (by automatically running the Experiment using several configurations to analyze the outputs), a first approach is to run several simulations from the same experiment, considering each simulation as an agent. A second approach, much more efficient for larger explorations, is to run an other type of experiment : the **Batch Experiment**.

We will start this part by learning how to **run several simulations** from the same experiment. Then, we will see how **batch experiments** work, and we will focus on how to use those batch experiments to explore models by using **exploration methods**.

Chapter 40

Run Several Simulations

To explore a model, the easiest and the most intuitive way to proceed is running several simulations with several parameter value, and see the differences from the output. GAMA provides you the possibility to launch several simulations from the GUI.

Index

Create a simulation

Let's remind you that in GAMA, everything is an **agent**. We already saw that the **"world" agent** is the **agent of the model**. The model is thus a **species**, called modelName_model :

```
model toto // <- the name of the species is "toto_model"
```

New highlight of the day : an **Experiment** is also an agent ! It's a special agent which will instantiate automatically an agent from the model species. You can then perfectly create agents (*model* agents) from your experiment, using the statement `create` :

```
model multi_simulations // the "world" is an instance of the "
   multi_simulations_model"

global {
}

experiment my_experiment type:gui  {
    init {
        create multi_simulations_model;
    }
}
```

This sort model will instantiate 2 simulations (two instance of the model) : one is created automatically by the experiment, and the second one is explicitly created through the statement `create`.

To simplify the syntax, you can use the built-in attribute `simulation` of your **experiment**. When you have a model called "multi_simulations", the two following lines are strictly equal :

```
create multi_simulations_model;
create simulation;
```

As it was the case for creating regular species, you can specify the parameters of your agent during the creation through the facet `with:` :

```
model multi_simulations

global {
    rgb bgd_color;
}

experiment my_experiment type:gui  {
    parameter name:"background color:" var:bgd_color init:#blue;
    init {
        create simulation with:[bgd_color::#red];
    }
    output {
        display "my_display" background:bgd_color{}
    }
}
```

Manipulate simulations

When you think the simulations as agents, it gives you a lot of new possibilities. You can for example create a reflex from your experiment, asking to create simulations **during the experiment execution** !

The following short model for example will create a new simulation at each 10 cycles :

```
model multi_simulations

global {
    init {
        write "new simulation created ! Its name is "+name;
    }
}

experiment my_experiment type:gui  {
    init {
    }
    reflex when:(mod(cycle,10)=0 and cycle!=0) {
        create simulation;
    }
    output {
    }
}
```

You may ask, what is the purpose of such a thing ? Well, with such a short model, it is not very interesting, for sure. But you can imagine running a simulation, and if the simulation reaches a certain state, it can be closed, and another simulation can be run instead with different parameters (a simulation can be closed by doing a "do die" on itself). You can also imagine to run two simulations, and to communicate from one to an other through the experiment, as it is shown in this easy model, where agents can move from one simulation to another :

```
model smallWorld
```

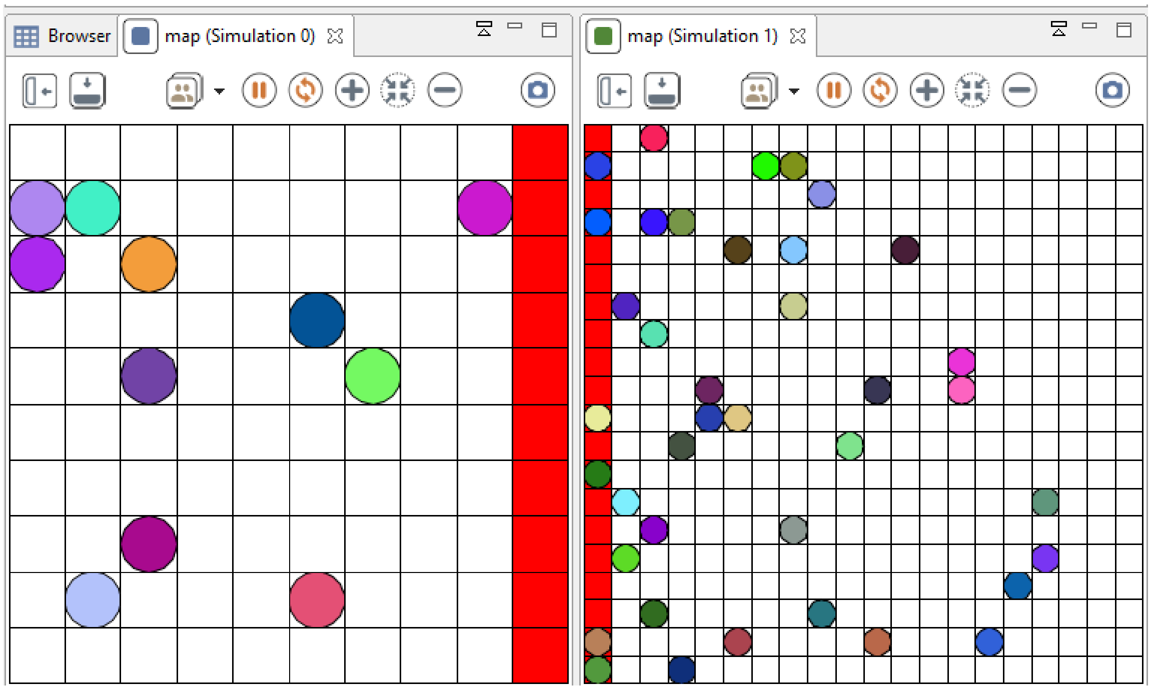
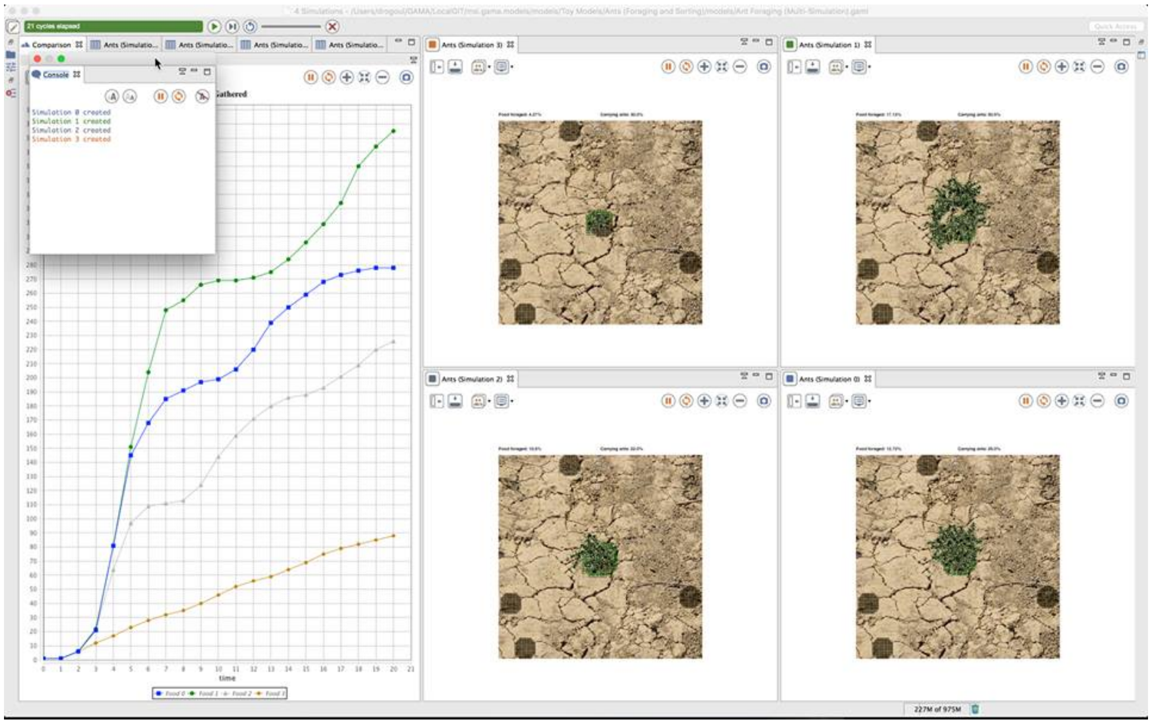


Figure 40.1: resources/images/exploringModel/change_world.png

```
global {
    int grid_size <- 10;
    bool modelleft <- true;
    int id<- 0;
    int nb_agents <- 50;

    init {
        create people number: nb_agents {
            my_cell <- one_of(cell);
            location <- my_cell.location;
        }
        if (modelleft) {
            ask cell where (each.grid_x = (grid_size - 1))  {
                color <- #red;
            }
        } else {
            ask cell where (each.grid_x = 0)  {
                color <- #red;
            }
        }
    }

    action changeWorld(rgb color, point loc) {
        create people with:[color::color, location::loc] {
            my_cell <- cell(location);
        }
    }
}

species people {
    rgb color <- rnd_color(255);
    cell my_cell;

    reflex move {
        if (modelleft and my_cell.color = #red) {
            ask smallWorld_model[1] {
                do changeWorld(myself.color, {100 - myself.
   location.x,myself.location.y});
            }
            do die;
        } else {
            list<cell> free_cells <- list<cell> (my_cell.
   neighbors) where empty(people inside each);
            if not empty(free_cells) {
                my_cell <- one_of(free_cells);
                location <- my_cell.location;
            }
        }

    }
    aspect default {
        draw circle(50/grid_size) color: color;
    }
}

grid cell width: grid_size height: grid_size;

experiment fromWorldToWorld type: gui {
    init {
         create simulation with:[grid_size::20, modelleft::false,
    id::1, nb_agents::0];
    }

    output {
        display map {
            grid cell lines: #black;
            species people;
        }
    }
}
```

Here is an other example of application of application, available in the model library. Here we run 4 times the Ant Foraging model, with different parameters.



```
Simulation 0 created
Simulation 1 created
Simulation 2 created
Simulation 3 created
```

Figure 40.2: resources/images/exploringModel/multi_foraging.jpg

Random seed

Defining the seed from the model

If you run several simulations, you may want to use the same seed for each one of those simulations (to compare the influence of a certain parameter, in exactly the same conditions).

Let's remind you that seed is a built-in attribute of the model. You than just need to specify the value of your seed during the creation of the simulation if you want to fix the seed :

```
create simulation with:[seed::10.0];
```

You can also specify the seed if you are inside the init scope of your global agent.

```
global {
    init {
        seed<-10.0;
    }
}
```

Notice that if you affect the value of your seed built-in directly in the global scope, the affectation of the parameters (for instance specified with the facet with of the statement create), and the "init" will be done after will be done at the end.

Defining the seed from the experiment

The experiment agent also have a built-in attribute seed. The value of this seed is defined in your simulation preferences. The first simulation created is created **with the seed value of the experiment**.

The following sequence diagram can explain you better how the affectation of the seed attribute works :

The affectation of an attribute is always done in this order : (1) the attribute is affected with a specific value in the species scope. If no attribute value is specified, the value is a default value. (2) if a value is specified for this attribute in the create statement, then the attribute value is affected again. (3) the attribute value can be changed again in the init scope.

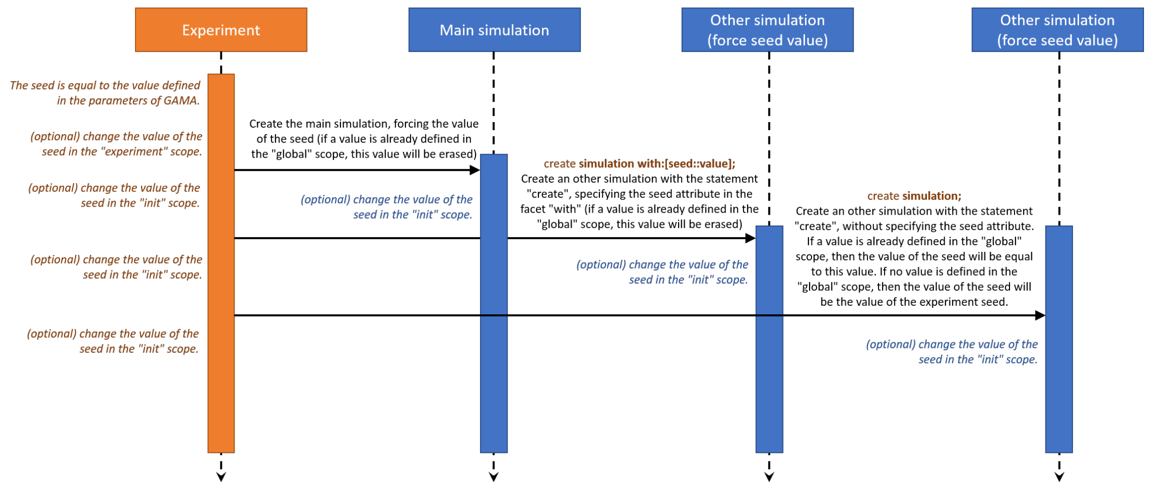

Experiment
Main simulation
Other simulation (force seed value)
Other simulation (force seed value)
The seed is equal to the value defined in the parameters of GAMA.
(optional) change the value of the seed in the "experiment" scope.
(optional) change the value of the seed in the "init" scope.
(optional) change the value of the seed in the "init" scope.
(optional) change the value of the seed in the "init" scope.
Create the main simulation, forcing the value of the seed (if a value is already defined in the "global" scope, this value will be erased)
(optional) change the value of the seed in the "init" scope.
create simulation with:[seed::value];
Create an other simulation with the statement "create", specifying the seed attribute in the facet "with" (if a value is already defined in the "global" scope, this value will be erased)
(optional) change the value of the seed in the "init" scope.
create simulation;
Create an other simulation with the statement "create", without specifying the seed attribute. If a value is already defined in the "global" scope, then the value of the seed will be equal to this value. If no value is defined in the "global" scope, then the value of the seed will be the value of the experiment seed.
(optional) change the value of the seed in the "init" scope.

Figure 40.3: resources/images/exploringModel/sequence_diagram_seed_affectation.png

Run several simulations with the same random numbers

The following code shows how to run several simulations with a specific seed, determined from the experiment agent :

```
model multi_simulations

global {
    init {
        create my_species;
    }
}

species my_species skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#green;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
    float seed <- seedValue; // force the value of the seed.
    init {
        // create a second simulation with the same seed as the
main simulation
        create simulation with:[seed::seedValue];
    }
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

When you run this simulation, their execution is exactly similar.

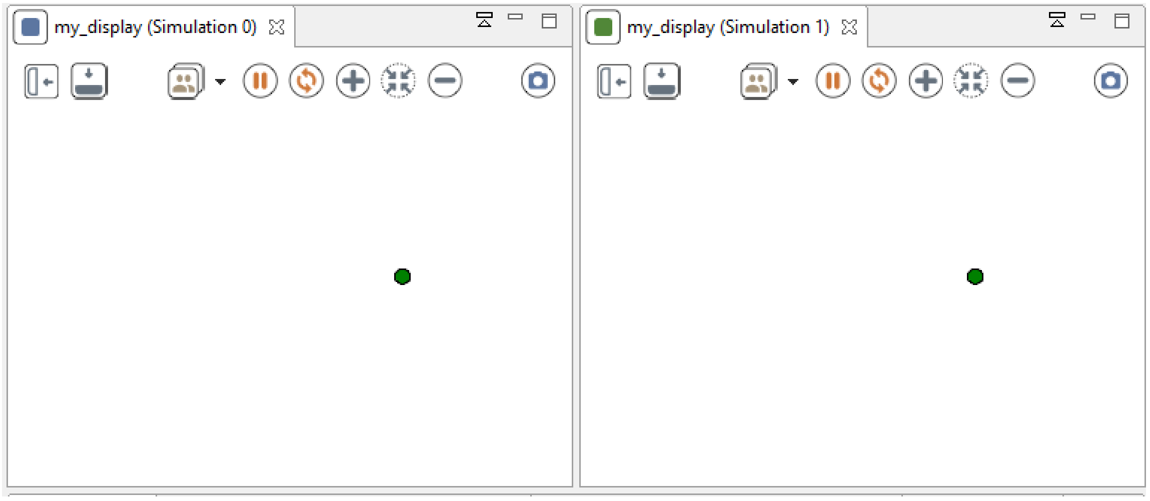


Figure 40.4: resources/images/exploringModel/same_simulation_one_agent.png

Let's try now to add a new species in this model, and to add a parameter to the simulation for the number of agents created for this species.

```
model multi_simulations

global {
    int number_of_speciesB <- 1;
    init {
        create my_speciesA;
        create my_speciesB number:number_of_speciesB;
    }
}

species my_speciesA skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#green;
    }
}

species my_speciesB skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#red;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
    float seed <- seedValue; // force the value of the seed.
    init {
        create simulation with:[seed::seedValue,
   number_of_speciesB::2];
    }
    output {
        display my_display {
            species my_speciesA aspect:base;
            species my_speciesB aspect:base;
        }
    }
}
```

Then you run the experiment, you may find something strange...

Even if the first step seems ok (the greed agent and one of the two red agent is initialized with the same location), the simulation differs completly. You should have expected to have

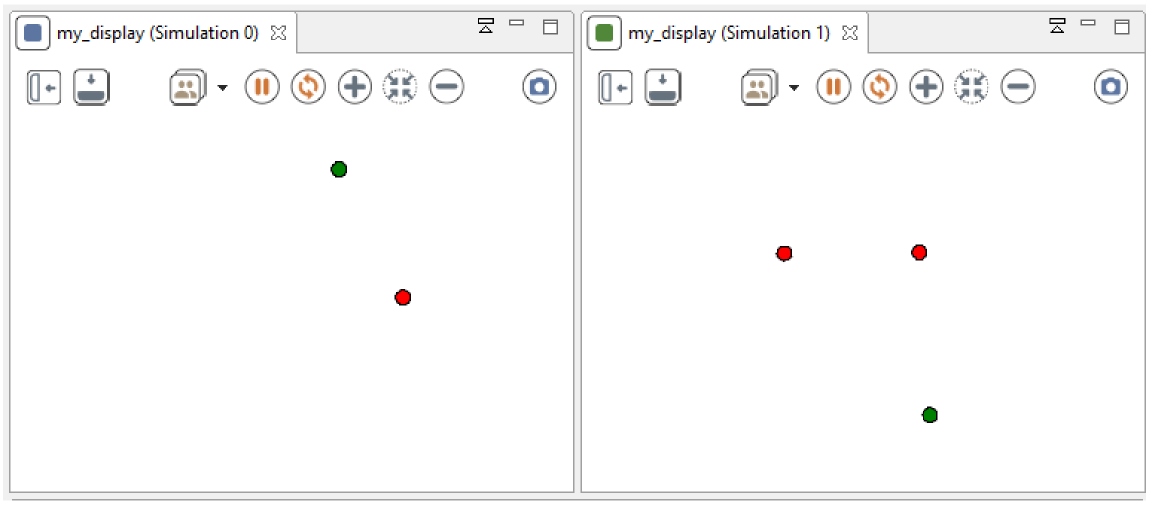


Figure 40.5: resources/images/exploringModel/same_simulation_2_species.png

the same behavior for the greed agent in both of the simulation, but it is not the case. The explaination of this behavior is that a random number generator has generated more random numbers in the second simulation than in the first one.

If you don't understand, here is a short example that may help you to understand better :

```
model multi_simulations

global {
    int iteration_number <- 1;
    reflex update {
        float value;
        loop times:iteration_number {
            value<-rnd(10.0);
            write value;
        }
        write "cycle "+cycle+" in experiment "+name+" : "+value;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
    float seed <- seedValue; // force the value of the seed.
    init {
        create simulation with:[seed::seedValue,iteration_number
   ::2];
    }
    output {
    }
}
```

The output will be something like that :

```
7.67003069780383
cycle 0 in experiment multi_simulations_model0 : 7.67003069780383
7.67003069780383
0.22889843360303863
cycle 0 in experiment multi_simulations_model1 :
   0.22889843360303863
0.22889843360303863
cycle 1 in experiment multi_simulations_model0 :
   0.22889843360303863
4.5220913306263855
0.8363180333035425
cycle 1 in experiment multi_simulations_model1 :
   0.8363180333035425
4.5220913306263855
cycle 2 in experiment multi_simulations_model0 :
   4.5220913306263855
5.460148568140819
4.158355846617511
cycle 2 in experiment multi_simulations_model1 :
   4.158355846617511
0.8363180333035425
cycle 3 in experiment multi_simulations_model0 :
   0.8363180333035425
1.886091659169562
4.371253083874633
cycle 3 in experiment multi_simulations_model1 :
   4.371253083874633
```

Which means :

Cycle	Value generated in simulation 0	Value generated in simulation 1
1	7.67003069780383	7.67003069780383

Cycle	Value generated in simulation 0	Value generated in simulation 1
2	0.228898433603038630	0.228898433603038630 4.5220913306263855 0.83631803330354250
3	4.5220913306263855	5.460148568140819 4.158355846617511

When writing your models, you have to be aware of this behavior. Remember that each simulation has it's own random number generator.

Change the RNG

The RNG (random number generator) can also be changed : `rng` is a string built-in attribute of the experiment (and also of the model). You can choose among the following rng : - mersenne (by default) - cellular - java

The following model shows how to run 4 simulations with the same seed but with some different RNG :

```
model multi_simulations

global {
    init {
        create my_species number:50;
    }
}

species my_species skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw square(2) color:#blue;
    }
}

experiment my_experiment type:gui  {
    float seed <- 10.0;
    init {
        create simulation with:[rng::"cellular",seed::10.0];
        create simulation with:[rng::"java",seed::10.0];
    }
    output {
        display my_display {
            species my_species aspect:base;
            graphics "my_graphic" {
                draw rectangle(35,10) at:{0,0} color:#lightgrey;
                draw rng at:{3,3} font:font("Helvetica", 20 , #
   plain) color:#black;
            }
        }
    }
}
```

Chapter 41

Defining Batch Experiments

Batch experiments allows to execute numerous successive simulation runs.They are used to explore the parameter space of a model or to optimize a set of model parameters.

A Batch experiment is defined by:

```
experiment exp_title type: batch {
   [parameter to explore]
   [exploration method]
   [reflex]
   [permanent]
}
```

Table of contents

The batch experiment facets

Batch experiment have the following three facets: * until: (expression) Specifies when to stop each simulations. Its value is a condition on variables defined in the model. The run will stop when the condition is evaluated to true. If omitted, the first simulation run will go forever, preventing any subsequent run to take place (unless a halt command is used in the model itself). * repeat: (integer) A parameter configuration corresponds to a set of values assigned to each parameter. The attribute repeat specifies the number of times each configuration will be repeated, meaning that as many simulations will be run with the same parameter values. Different random seeds are given to the pseudo-random number generator. This allows to get some statistical power from the experiments conducted. Default value is 1. * keep_seed: (boolean) If true, the same series of random seeds will be used from one parameter configuration to another. Default value is false.

```
experiment my_batch_experiment type: batch repeat: 5 keep_seed:
   true until: time = 300 {
   [parameter to explore]
   [exploration method]
}
```

Action *step The_step_action of an experiment is called at the end of a simulation. It is possible to override this action to apply a specific action at the end of each simulation. Note that at the experiment level, you have access to all the species and all the global variables.*

For instance, the following experiment runs the simulation 5 times, and, at the end of each simulation, saves the people agents in a shapefile.

```
experiment 'Run 5 simulations' type: batch repeat: 5 keep_seed:
   true until: ( time > 1000 ) {
    int cpt <- 0;
    action _step_ {
        save people type:"shp" to:"people_shape" + cpt + ".shp"
   with: [is_infected::"INFECTED",is_immune::"IMMUNE"];
        cpt <- cpt + 1;
    }
}

A second solution to achieve the same result is to use reflexes (
   see below).
```

Reflexes

It is possible to write reflexes inside a batch experiment. This reflex will be executed at the end of each simulation. For instance, the following reflex writes at the end of each simulation the value of the variable *food_gathered*:

```
reflex info_sim {
    write "Running a new simulation " + simulation + " -> " +
   food_gathered;
}
```

Permanent

The **permanent** section allows to define a output block that will not be re-initialized at the beginning of each simulation but will be filled at the end of each simulation. For instance, this **permanent** section will allows to display for each simulation the end value of the *food_-gathered* variable.

```
permanent {
    display Ants background: rgb('white') refresh:every(1) {
        chart "Food Gathered" type: series {
            data "Food" value: food_gathered;
        }
    }
}
```

Chapter 42

Exploration Methods

Several batch methods are currently available. Each is described below.

Table of contents

The method element

The optional method element controls the algorithm which drives the batch.

If this element is omitted, the batch will run in a classical way, changing one parameter value at each step until all the possible combinations of parameter values have been covered. See the Exhaustive exploration of the parameter space for more details.

When used, this element must contain at least a name attribute to specify the algorithm to use. It has theses facets: * minimize or a maximize (mandatory for optimization method): a attribute defining the expression to be optimized. * aggregation (optional): possible values ("min", "max"). Each combination of parameter values is tested **repeat** times. The aggregated fitness of one combination is by default the average of fitness values obtained with those repetitions. This facet can be used to tune this aggregation function and to choose to compute the aggregated fitness value as the minimum or the maximum of the obtained fitness values. * other parameters linked to exploration method (optional) : see below for a description of these parameters.

Exemples of use of the method elements:

```
method exhaustive minimize: nb_infected ;

method genetic pop_dim: 3 crossover_prob: 0.7 mutation_prob: 0.1
   nb_prelim_gen: 1 max_gen: 5  minimize: nb_infected aggregation
   : "max";
```

Exhaustive exploration of the parameter space

Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way.

Example (models/ants/batch/ant_exhaustive_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
}
```

The order of the simulations depends on the order of the param. In our example, the first combinations will be the followings: * evaporation_rate = 0.1, diffusion_rate = 0.1, (2 times) * evaporation_rate = 0.1, diffusion_rate = 0.4, (2 times) * evaporation_rate = 0.1, diffusion_-rate = 0.7, (2 times) * evaporation_rate = 0.1, diffusion_rate = 1.0, (2 times) * evaporation_-rate = 0.2, diffusion_rate = 0.1, (2 times) * ...

Note: this method can also be used for optimization by adding an method element with maximize or a minimize attribute:

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method exhaustive maximize: food_gathered;
}
```

Hill Climbing

Name: hill_climbing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article.

Algorithm:

```
 Initialization of an initial solution s
 iter = 0
 While iter <= iter_max, do:
   Choice of the solution s' in the neighborhood of s that
   maximize the fitness function
   If f(s') > f(s)
     s = 's
   Else
     end of the search process
   EndIf
   iter = iter + 1
 EndWhile
```

Method parameters: * iter_max: number of iterations

Example (models/ants/batch/ant_hill_climbing_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method hill_climbing iter_max: 50 maximize : food_gathered;
}
```

Simulated Annealing

Name: annealing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article.

Algorithm:

```
Initialization of an initial solution s
temp = temp_init
While temp > temp_end, do:
  iter = 0
  While iter < nb_iter_cst_temp, do:
    Random choice of a solution s2 in the neighborhood of s
    df = f(s2)-f(s)
    If df > 0
      s = s2
    Else,
      rand = random number between 0 and 1
      If rand < exp(df/T)
        s = s2
      EndIf
    EndIf
    iter = iter + 1
  EndWhile
  temp = temp * nb_iter_cst_temp
EndWhile
```

Method parameters: * temp_init: Initial temperature * temp_end: Final temperature * temp_decrease: Temperature decrease coefficient * nb_iter_cst_temp: Number of iterations per level of temperature

Example (models/ants/batch/ant_simulated_annealing_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method annealing temp_init: 100  temp_end: 1 temp_decrease:
   0.5 nb_iter_cst_temp: 5 maximize: food_gathered;
}
```

Tabu Search

Name: tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article.

Algorithm:

```
 Initialization of an initial solution s
 tabuList = {}
 iter = 0
 While iter <= iter_max, do:
   Choice of the solution s2 in the neighborhood of s such that:
     s2 is not in tabuList
     the fitness function is maximal for s2
   s = s2
   If size of tabuList = tabu_list_size
     removing of the oldest solution in tabuList
   EndIf
   tabuList = tabuList + s
   iter = iter + 1
 EndWhile
```

Method parameters: * iter_max: number of iterations * tabu_list_size: size of the tabu list

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method tabu iter_max: 50 tabu_list_size: 5 maximize:
   food_gathered;
}
```

Reactive Tabu Search

Name: reactive_tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle.

Method parameters: * iter_max: number of iterations * tabu_list_size_init: initial size of the tabu list * tabu_list_size_min: minimal size of the tabu list * tabu_list_size_max: maximal size of the tabu list * nb_tests_wthout_col_max: number of movements without collision before shortening the tabu list * cycle_size_min: minimal size of the considered cycles * cycle_size_max: maximal size of the considered cycles

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
```

```
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method reactive_tabu iter_max: 50 tabu_list_size_init: 5
   tabu_list_size_min: 2 tabu_list_size_max: 10
   nb_tests_wthout_col_max: 20 cycle_size_min: 2 cycle_size_max:
   20 maximize: food_gathered;
}
```

Genetic Algorithm

Name: genetic Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article. The principle of GA is to search an optimal solution by applying evolution operators on an initial population of solutions There are three types of evolution operators: * Crossover: Two solutions are combined in order to produce new solutions * Mutation: a solution is modified * Selection: only a part of the population is kept. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Representation of the solutions: * Individual solution: {Param1 = val1; Param2 = val2; …} * Gene: Parami = vali

Initial population building: the system builds nb_prelim_gen random initial populations composed of pop_dim individual solutions. Then, the best pop_dim solutions are selected to be part of the initial population.

Selection operator: roulette-wheel selection: the probability to choose a solution is equals to: fitness(solution)/ Sum of the population fitness. A solution can be selected several times. Ex: population composed of 3 solutions with fitness (that we want to maximize) 1, 4 and 5. Their probability to be chosen is equals to 0.1, 0.4 and 0.5.

Mutation operator: The value of one parameter is modified. Ex: The solution {Param1 = 3; Param2 = 2} can mute to {Param1 = 3; Param2 = 4}

Crossover operator: A cut point is randomly selected and two new solutions are built by taking the half of each parent solution. Ex: let {Param1 = 4; Param2 = 1} and {Param1 = 2; Param2 = 3} be two solutions. The crossover operator builds two new solutions: {Param1 = 2; Param2 = 1} and {Param1 = 4; Param2 = 3}.

Method parameters: * pop_dim: size of the population (number of individual solutions) * crossover_prob: crossover probability between two individual solutions * mutation_prob: mutation probability for an individual solution * nb_prelim_gen: number of random populations used to build the initial population * max_gen: number of generations

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 ,
    0.2 , 0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means
   100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0
   unit: 'rate every cycle (1.0 means 100%)' step: 0.3;
    method genetic maximize: food_gathered pop_dim: 5
   crossover_prob: 0.7 mutation_prob: 0.1 nb_prelim_gen: 1
   max_gen: 20;
}
```

Optimizing Models

Now you are becoming more comfortable with GAML, it is time to think about how the runtime works, to be able to run some more optimized models. Indeed, if you already tried to write some models by yourself using GAML, you could have notice that the execution time depends a lot of how you implemented your model !

We will first present you in this part some **runtime concepts** (and present you the species facet `scheduler`), and we will then show you some **tips to optimize your models** (how to increase performances using scheduler, grids, displays and how to choose your operators).

Chapter 43

Runtime Concepts

When a model is being simulated, a number of algorithms are applied, for instance to determine the order in which to run the different agents, or the order in which the initialization of agents is performed, etc. This section details some of them, which can be important when building models and understanding how they will be effectively simulated.

Table of contents

Simulation initialization

Once the user launches an experiment, GAMA starts the initialization of the simulation. First it creates a `world` agent.

It initializes all its attributes with their init values. This includes its shape (that will be used as environment of the simulation).

If a species of type grid exists in the model, agents of species are created.

Finally the init statement is executed. It should include the creation of all the other agents of regular species of the simulation. After their creation and initialization, they are added in the list members the world (that contains all the micro-agent of the world).

Agents Creation

Except grid agents, other agents are created using the create statement. It is used to allocate memory for each agent and to initialize all its attributes.

If no explicit initialization exists for an attribute, it will get the default value corresponding to its type.

The initialization of an attribute can be located at several places in the code; they are executed in the following order (which means that, if several ways are used, the attribute will finally have the value of the last applied one): * using the from: facet of the create statement; * in the embedded block of the create statement; * in the attribute declaration, using the init facet; * in the init block of the species.

Agents Step

When an agent is asked to *step*, it means that it is expected to update its variables, run its behaviors and then *step* its micro-agents (if any).

```
step of agent agent_a
    {
        species_a <- agent_a.species
        architecture_a <- species_a.architecture
        ask architecture_a to step agent_a {
             ask agent_a to update species_a.variables
             ask agent_a to run architecture_a.behaviors
        }

        ask each micro-population mp of agent_a to step {
            list<agent> sub-agents <- mp.
   compute_agents_to_schedule
            ask each agent_b of sub-agents to step //… recursive
   call...
        }
    }
```

Schedule Agents

The global scheduling of agents is then simply the application of this previous *step* to the *experiment agent*, keeping in mind that this agent has only one micro-population (of simulation agents, each instance of the model species), and that the simulation(s) inside this population contain(s), in turn, all the "regular" populations of agents of the model.

To influence this schedule, then, one possible way is to change the way populations compute their lists of agents to schedule, which can be done in a model by providing custom definitions to the "schedules:" facet of one or several species.

A practical application of this facet is to reduce simulation artifacts created by the default scheduling of populations, which is sequential (i.e. their agents are executed in turn in their order of creation). To enable a pseudo-parallel scheduling based on a random scheduling recomputed at each step, one has simply to define the corresponding species like in the following example:

```
species A schedules: shuffle(A) …{}
```

Moving further, it is possible to enable a completely random scheduling that will eliminate the sequential scheduling of populations:

```
global schedules: [world] + shuffle(A + B + C) …{}

species A schedules: [] …{}
species B schedules: [] …{}
species C schedules: [] …{}
```

It is important to (1) explicitly invoke the scheduling of the world (although it doesn't have to be the first); (2) suppress the population-based scheduling to avoid having agent being scheduled 2 times (one time in the custom definition, one time by their population).

Other schemes are possible. For instance, the following definition will completely suppress the default scheduling mechanism to replace it with a custom scheduler that will execute the world, then all agents of species A in a random way and then all agents of species B in their order of creation:

```
global schedules: [world] + shuffle(A) + B …{} // explicit
   scheduling in the world

species A schedules [];

species B schedules: [];
```

Complex conditions can be used to express which agents need to be scheduled each step. For instance, in the following definition, only agents of A that return true to a particular condition are scheduled:

```
species A schedules: A where each.can_be_scheduled() {

    bool can_be_scheduled() {
         …
         returns true_or_false;
    }
}
```

Be aware that enabling a custom scheduling can potentially end up in non-functional simulations. For example, the following definitions will result in a simulation that will **never be executed**:

```
global schedules: [] {}; // the world is NEVER scheduled

species my_scheduler schedules: [world] ; // so its micro-species
    'my_scheduler' is NOT scheduled either.
```

and this one will result in an **infinite loop** (which will trigger a stack overflow at some point):

```
global {} // The world is normally scheduled...

species my_scheduler schedules: [world]; // … but schedules
   itself again as a consequence of scheduling the micro-species
   'my_scheduler'
```

Chapter 44

Optimizing Models

This page aims at presenting some tips to optimize the memory footprint or the execution time of a model in GAMA.

Note: since GAMA 1.6.1, some optimizations have become obsolete because they have been included in the compiler. They have, then, been removed from this page. For instance, writing 'rgb(0,0,0)' is now compiled directly as "°black'.

Table of contents

machine_time

In order to optimize a model, it is important to exactly know which part of the model take times. The simplest to do that is to use the **machine_time** built-in global variable that gives the current time in milliseconds. Then to compute the time taken by a statement, a possible way is to write:

```
float t <- machine_time;
// here a block of instructions that you consider as "critical"
// ...
write "duration of the last instructions: " + (machine_time - t);
```

Scheduling

If you have a species of agents that, once created, are not supposed to do anything more (i.e. no behavior, no reflex, their actions triggered by other agents, their attributes being simply read and written by other agents), such as a "data" grid, or agents representing a "background" (from a shape file, etc.), consider using the `schedules: []` facet on the definition of their species. This trick allows to tell GAMA to not schedule any of these agents.

```
grid my_grid height: 100 width: 100 schedules: []
{
	...
}
```

The `schedules:` facet is dynamically computed (even if the agents are not scheduled), so, if you happen to define agents that only need to be scheduled every x cycles, or depending on a condition, you can also write `schedules:` to implement this. For instance, the following species will see its instances scheduled every 10 steps and only if a certain condition is met:

```
species my_species schedules: (every 10) ? (condition ?
   my_species : []) : []
{
     …
}
```

In the same way, modelers can use the frequency facet to define when the agents of a species are going to be activated. By setting this facet to 0, the agents are never activated.

```
species my_species frequency: 0
{
     ...
}
```

Grid

Optimization Facets

In this section, we present some facets that allow to optimize the use of grid (in particular in terms of memories). Note that all these facet can be combined (see the Life model from the Models library).

use_regular_agents

If false, then a special class of agents is used. This special class of agents used less memories but has some limitation: the agents cannot inherit from a "normal" species, they cannot have sub-populations, their name cannot be modified, etc.

```
grid cell width: 50 height: 50 use_regular_agents: false ;
```

use_individual_shapes

If false, then only one geometry is used for all agents. This facet allows to gain a lot of memory, but should not be used if the geometries of the agents are often activated (for instance, by an aspect).

```
grid cell width: 50 height: 50 use_individual_shapes: false ;
```

Operators

List operators

first_with

It is sometimes necessary to randomly select an element of a list that verifies a certain condition. Many modelers use the **one_of** and the **where** operators to do this:

```
bug one_big_bug <- one_of (bug where (each.size > 10));
```

Whereas it is often more optimized to use the **shuffle** operator to shuffle the list, then the **first_with** operator to select the first element that verifies the condition:

```
bug one_big_bug <- shuffle(bug) first_with (each.size > 10);
```

where / count

It is quite common to want to count the number of elements of a list or a container that verify a condition. The obvious to do it is :

```
int n <- length(my_container where (each.size > 10));
```

This will however create an intermediary list before counting it, and this operation can be time consuming if the number of elements is important. To alleviate this problem, GAMA includes an operator called **count** that will count the elements that verify the condition by iterating directly on the container (no useless list created) :

```
int n <- my_container count (each.size > 10);
```

Spatial operators

container of agents in closest_to, at_distance, overlapping, inside

Several spatial query operators (such as **closest_to**, **at_distance**, **overlapping** or **inside**) allow to restrict the agents being queried to a container of agents. For instance, one can write:

```
agent closest_agent <- a_container_containing_agents closest_to
   self;
```

This expression is formally equivalent to :

```
agent closest_agent <- a_container_containing_agent with_min_of (
   each distance_to self);
```

But it is much faster **if your container is large**, as it will query the agents using a spatial index (instead of browsing through the whole container). Note that in some cases, when you have a small number of agents, the first syntax will be faster. The same applies for the other operators.

Now consider a very common case: you need to restrict the agents being queried, not to a container, but to a species (which, actually, acts as a container in most cases). For instance, you want to know which predator is the closest to the current agent. If we apply the pattern above, we would write:

```
predator closest_predator <- predator with_min_of (each
   distance_to self);
```

or

```
predator closest_predator <- list(predator) closest_to self;
```

But these two operators can be painfully slow if your species has many instances (even in the second form). In that case, always prefer using **directly** the species as the left member:

```
predator closest_ predator <- predator closest_to self;
```

Not only is the syntax clearer, but the speed gain can be phenomenal because, in that case, the list of instances is not used (we just check if the agent is an instance of the left species).

However, what happens if one wants to query instances belonging to 2 or more species ? If we follow our reasoning, the immediate way to write it would be (if predator 1 and predator 2 are two species):

```
agent closest_agent <- (list(predator1) + list(predator2))
   closest_to self;
```

or, more simply:

```
agent closest_agent <- (predator1 + predator2) closest_to self;
```

The first syntax suffers from the same problem than the previous syntax: GAMA has to browse through the list (created by the concatenation of the species populations) to filter agents. The solution, then, is again to use directly the species, as GAMA is clever enough to create a temporary “fake” population out of the concatenation of several species, which can be used exactly like a list of agents, but provides the advantages of a species population (no iteration made during filtering).

Accelerate closest_to with a first spatial filtering

The **closest_to** operator can sometimes be slow if numerous agents are concerned by this query. If the modeler is just interested by a small subset of agents, it is possible to apply a first spatial filtering on the agent list by using the **at_distance** operator. For example, if the modeler wants first to do a spatial filtering of 10m:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self
   ;
```

To be sure to find an agent, the modeler can use a test statement:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self
   ;
if (closest_agent = nil) {closest_agent  <- predator1 closest_to
   self;}
```

Displays

shape

It is quite common to want to display an agent as a circle or a square. A common mistake is to mix up the shape to draw and the geometry of the agent in the model. If the modeler just wants to display a particular shape, he/she should not modify the agent geometry (which is a point by default), but just specify the shape to draw in the agent aspect.

```
species bug {
     int size <- rnd(100);

      aspect circle {
          draw circle(2) color: °blue;
      }
}
```

circle vs square / sphere vs cube

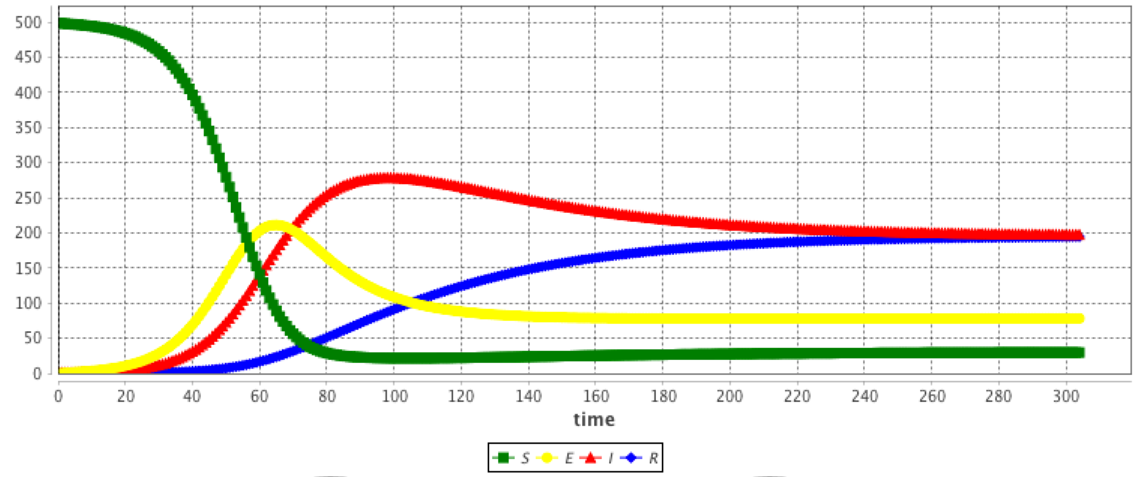
Note that in OpenGL and Java2D (the two rendering subsystems used in GAMA), creating and drawing a circle geometry is more time consuming than creating and drawing a square (or a rectangle). In the same way, drawing a sphere is more time consuming than drawing a cube. Hence, if you want to optimize your model displays and if the rendering does not explicitly need "rounded" agents, try to use squares/cubes rather than circles/spheres.

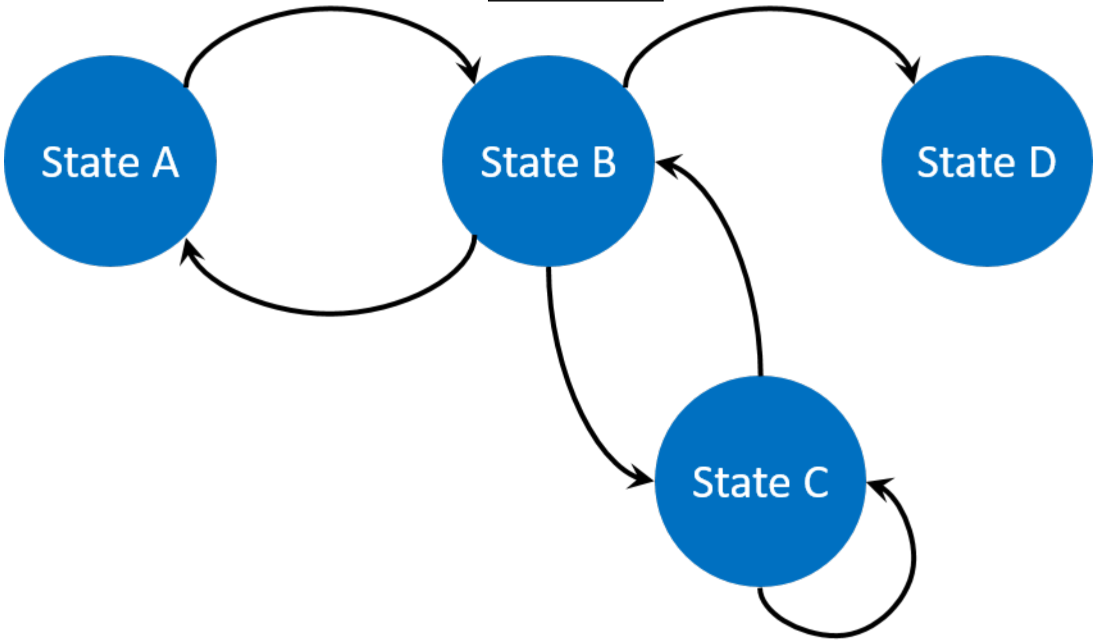
OpenGL refresh facets

In OpenGL display, it is possible to specify that it is not necessary to refresh a layer with the facet **refresh**. If a species of agents is never modified in terms of visualization (location, shape or color), you can set **refresh** to false. Example:

```
display city_display_opengl type: opengl{
     species building aspect: base refresh: false;
     species road aspect: base refresh: false;
     species people aspect: base;
}
```

Multi-Paradigm Modeling





Multi-paradigm modeling is a research field focused on how to define a model semantically. From the beginning of this step by step tutorial, our approach is based on behavior (or reflex), for each agents. In this part, we will see that GAMA provides other ways to implement your model, using several control architectures. Sometime, it will be easier to implement your models choosing other paradigms.

In a first part, we will see how to use some **control architectures** which already exist in GAML, such as finite state machine architecture, task based architecture or user control architecture. In a second part, we will see an other approach, a math approach, through **equations**.

Chapter 45

Control Architectures

GAMA allows to attach built-in control architecture to agents.

These control architectures will give the possibility to the modeler to use for a species a specific control architecture in addition to the common behavior structure. Note that only one control architecture can be used per species.

The attachment of a control architecture to a species is done through the facets `control`.

For example, the given code allows to attach the `fsm` control architecture to the dummy species.

```
species dummy control: fsm {
}
```

GAMA integrates several agent control architectures that can be used in addition to the common behavior structure:

- fsm: finite state machine based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is in one single state. Such an agent needs to have one initial state (the state in which it will be at its initialization)
- weighted_tasks: task-based control architecture. At any given time, only the task only the task with the maximal weight is executed.
- sorted_tasks: task-based control architecture. At any given time, the tasks are all executed in the order specified by their weights (highest first).
- probabilistic_tasks: task-based control architecture. This architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen at each step.

- user_only: allows users to take control over an agent during the course of the simulation. With this architecture, only the user control the agents (no reflexes).
- user_first: allows users to take control over an agent during the course of the simulation. With this architecture, the user actions are executed before the agent reflexes.
- user_last: allows users to take control over an agent during the course of the simulation. With this architecture, the user actions are executed after the agent reflexes.

Index

Finite State Machine

FSM (Finite State Machine) is a finite state machine based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is in one single state. Such an agent needs to have one initial state (the state in which it will be at its initialization).

At each time step, the agent will:

- first (only if he just entered in its current state) execute statement embedded in the `enter` statement,
- then all the statements in the state statement
- it will evaluate the condition of each embedded transition statements. If one condition is fulfilled, the agent execute the embedded statements

Note that an agent executes only one state at each step.

Declaration

Using the FSM architecture for a species require to use the **control** facet:

```
species dummy control: fsm {
   ...
}
```

State

Attributes

- initial: a boolean expression, indicates the initial state of agent.
- final: a boolean expression, indicates the final state of agent.

Sub Statements

- enter: a sequence of statements to execute upon entering the state.
- exit: a sequence of statements to execute right before exiting the state. Note that the `exit` statement will be executed even if the fired transition points to the same state (the FSM architecture in GAMA does not implement 'internal transitions' like the ones found in UML state charts: all transitions, even "self-transitions", follow the same rules).
- transition: allows to define a condition that, when evaluated to true, will designate the next state of the life cycle. Note that the evaluation of transitions is short-circuited: the first one that evaluates to true, in the order in which they have been defined, will be followed. I.e., if two transitions evaluate to true during the same time step, only the first one will be triggered.

Things worth to be mentioned regarding these sub-statements:

- Obviously, only one definition of exit and enter is accepted in a given state
- Transition statements written in the middle of the state statements will only be evaluated at the end, so, even if it evaluates to true, the remaining of the statements found after the definition of the transition will be nevertheless executed. So, despite the appearance, a transition written somewhere in the sequence will "not stop" the state at that point (but only at the end).

Definition

A state can contain several statements that will be executed, at each time step, by the agent. There are three exceptions to this rule:

1. statements enclosed in `enter` will only be executed when the state is entered (after a transition, or because it is the initial state).
2. Those enclosed in `exit` will be executed when leaving the state as a result of a successful transition (and before the statements enclosed in the transition).
3. Those enclosed in a transition will be executed when performing this transition (but after the `exit` sequence has been executed).

For example, consider the following example:

```
species dummy control: fsm {
    state state1 initial: true {
        write string(cycle) + ":" + name + "->" + "state1";
        transition to: state2 when: flip(0.5) {
            write string(cycle) + ":" + name + "->" + "transition
    to state1";
        }
        transition to: state3 when: flip(0.2) ;
    }

    state state2 {
        write string(cycle) + ":" + name + "->" + "state2";
        transition to: state1 when: flip(0.5) {
            write string(cycle) + ":" + name + "->" + "transition
    to state1";
        }
        exit {
            write string(cycle) + ":" + name + "->" + "leave
   state2";
        }
    }

    state state3 {
        write string(cycle) + ":" + name + "->" + "state3";
        transition to: state1 when: flip(0.5)  {
            write string(cycle) + ":" + name + "->" + "transition
    to state1";
```

```
        }
        transition to: state2 when: flip(0.2)  ;
    }
}
```

the dummy agents start at *state1*. At each simulation step they have a probability of 0.5 to change their state to *state2*. If they do not change their state to *state2*, they have a probability of 0.2 to change their state to *state3*. In *state2*, at each simulation step, they have a probability of 0.5 to change their state to *state1*. At last, in *step3*, at each simulation step they have a probability of 0.5 to change their state to *state1*. If they do not change their state to *state1*, they have a probability of 0.2 to change their state to *state2*.

Here a possible result that can be obtained with one dummy agent:

```
0:dummy0->state1
0:dummy0->transition to state1
1:dummy0->state2
2:dummy0->state2
2:dummy0->leave state2
2:dummy0->transition to state1
3:dummy0->state1
3:dummy0->transition to state1
4:dummy0->state2
5:dummy0->state2
5:dummy0->leave state2
5:dummy0->transition to state1
6:dummy0->state1
7:dummy0->state3
8:dummy0->state2
```

Task Based

GAMA integrated several **task-based** control architectures. Species can define any number of tasks within their body. At any given time, only one or several tasks are executed according to the architecture chosen:

- **weighted_tasks** : in this architecture, only the task with the maximal weight is executed.

- **sorted_tasks** : in this architecture, the tasks are all executed in the order specified by their weights (biggest first)
- **probabilistic_tasks**: this architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen each step.

Declaration

Using the Task architectures for a species require to use the **control** facet:

```
species dummy control: weighted_tasks {
    ...
}
```

```
species dummy control: sorted_tasks {
    ...
}
```

```
species dummy control: probabilistic_tasks {
    ...
}
```

Task

Sub elements

Besides a sequence of statements like reflex, a task contains the following sub elements: * weight: Mandatory. The priority level of the task.

Definition

As reflex, a task is a sequence of statements that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler chooses a task to execute based on its current priority weight value.

For example, consider the following example:

```
species dummy control: weighted_tasks {
    task task1 weight: cycle mod 3 {
        write string(cycle) + ":" + name + "->" + "task1";
    }
    task task2 weight: 2 {
        write string(cycle) + ":" + name + "->" + "task2";
    }
}
```

As the **weighted_tasks** control architecture was chosen, at each simulation step, the dummy agents execute only the task with the highest behavior. Thus, when *cycle modulo 3* is higher to 2, task1 is executed; when *cycle modulo 3* is lower than 2, task2 is executed. In case when *cycle modulo 3* is equal 2 (at cycle 2, 5, ...), the only the first task defined (here task1) is executed.

Here the result obtained with one dummy agent:

```
0:dummy0->task2
1:dummy0->task2
2:dummy0->task1
3:dummy0->task2
4:dummy0->task2
5:dummy0->task1
6:dummy0->task2
```

User Control Architecture

user_only, user_first, user_last

A specific type of control architecture has been introduced to allow users to take control over an agent during the course of the simulation. It can be invoked using three different keywords: `user_only`, `user_first`, `user_last`.

```
species user control: user_only {
   ...
}
```

If the control chosen is `user_first`, it means that the user controlled panel is opened first, and then the agent has a chance to run its "own" behaviors (reflexes, essentially, or "init" in the case of a "user_init" panel). If the control chosen is `user_last`, it is the contrary.

user_panel

This control architecture is a specialization of the Finite State Machine Architecture where the "behaviors" of agents can be defined by using new constructs called `user_panel` (and one `user_init`), mixed with "states" or "reflexes". This `user_panel` translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each `user_panel`, like a `state` in FSM, can have a `enter` and `exit` sections, but it is only defined in terms of a set of `user_command`s which describe the different action buttons present in the panel.

user_commands can also accept inputs, in order to create more interesting commands for the user. This uses the `user_input` statement (and not operator), which is basically the same as a temporary variable declaration whose value is asked to the user. Example:

As `user_panel` is a specialization of `state`, the modeler has the possibility to describe several panels and choose the one to open depending on some condition, using the same syntax than for finite state machines : * either adding `transitions` to the user_panels, * or setting the `state` attribute to a new value, from inside or from another agent.

This ensures a great flexibility for the design of the user interface proposed to the user, as it can be adapted to the different stages of the simulation, etc...

Follows a simple example, where, every 10 steps, and depending on the value of an attribute called "advanced", either the basic or the advanced panel is proposed.

```
species user control:user_only {
   user_panel default initial: true {
      transition to: "Basic Control" when: every (10) and !
   advanced_user_control;
      transition to: "Advanced Control" when: every(10) and
   advanced_user_control;
   }

   user_panel "Basic Control" {
      user_command "Kill one cell" {
         ask (one_of(cell)){
            do die;
         }
      }
      user_command "Create one cell" {
        create cell ;
      }
      transition to: default when: true;
   }
   user_panel "Advanced Control" {
      user_command "Kill cells" {
        user_input "Number" returns: number type: int <- 10;
        ask (number among cell){
           do die;
        }
      }
      user_command "Create cells" {
        user_input "Number" returns: number type: int <- 10;
        create cell number: number ;
      }
      transition to: default when: true;
   }
}
```

The panel marked with the "*initial: true*" facet will be the one run first when the agent is supposed to run. If none is marked, the first panel (in their definition order) is chosen.

A special panel called user_init will be invoked only once when initializing the agent if it is defined. If no panel is described or if all panels are empty (i.e. no user_commands), the control view is never invoked. If the control is said to be "user_only", the agent will then not run any of its behaviors.

user_controlled

Finally, each agent provided with this architecture inherits a boolean attribute called `user_controlled`. If this attribute becomes false, no panels will be displayed and the agent will run "normally" unless its species is defined with a `user_only` control. //: # (endConcept|user_control_architecture)

Other Control Architectures

Some other control architectures are available in additional plugins. For instance, BDI (Belief, desire, intention) architecture is available. Feel free to read about it if you want to learn more.

You need some other control architectures for your model ? Feel free to make your suggestion to the team of developer through the mailing list. Remember also that GAMA is an open-source platform, you can design your own control architecture easily. Go to the section Community/contribute if you want to jump into coding !

Chapter 46

Using Equations

Introduction

ODEs (Ordinary Differential Equations) are often used in ecology or in epidemiology to describe the macroscopic evolution over time of a population. Generally the whole population is split into several compartments. The state of the population is described by the number of individuals in each compartment. Each equation of the ODE system describes the evolution of the number of individual in a compartment. In such an approach individuals are not taken into account individually, with own features and behaviors. In contrary they are aggregated in a compartment and reduced to a number.

A classical example is the SIR epidemic model representing the spreading of a disease in a population. The population is split into 3 compartments: S (Susceptible), I (Infected), R (Recovered). (see below for the equation)

In general the ODE systems cannot be analytically solved, i.e. it is not possible to find the equation describing the evolution of the number of S, I or R. But these systems can be numerically integrated in order to get the evolution. A numerical integration computes step after step the value of S, I and R. Several integration methods exist (e.g. Euler, Runge-Kutta…), each of them being a compromise between accuracy and computation time. The length of the integration step has also a huge impact on precision. These models are deterministic.

This approach makes a lot of strong hypotheses. The model does not take into account space. The population is considered has infinite and homogeneously mixed, so that any agent can interact with any other one.

Example of a SIR model

In the SIR model, the population is split into 3 compartments: S (Susceptible), I (Infected), R (Recovered). This can be represented by the following Forrester diagram: boxes represent stocks (i.e. compartments) and arrows are flows. Arrows hold the rate of a compartment population flowing to another compartment.

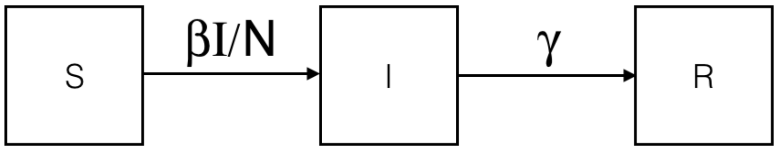


Figure 46.1: SIR-compartment.png

The corresponding ODE system contains one equation per stock. For example, the I compartment evolution is influenced by an inner (so positive) flow from the S compartment and an outer (so negative) flow to the R compartment.

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I \\ \dfrac{dR}{dt} = \gamma I \end{cases}$$

Figure 46.2: SIR-equations.png

Integrating this system using the Runge-Kutta 4 method provides the evolution of S, I and R over time. The initial values are: * S = 499 * I = 1 * R = 0 * beta = 0.4 * gamma = 0.1 * h = 0.1

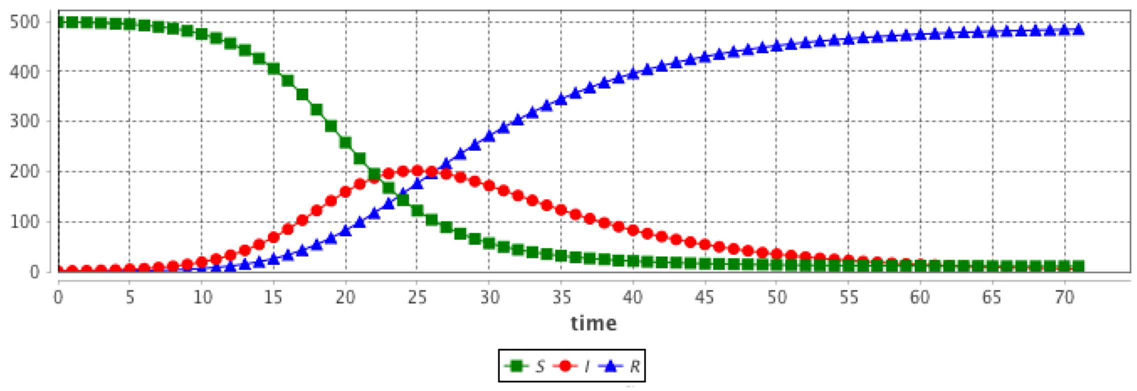


Figure 46.3: SIR-result.png

Why and when can we use ODE in agent-based models ?

These hypotheses are very strong and cannot be fulfilled in agent-based models.

But in some multi-scale models, some entities can be close. For example if we want to implement a model describing the worldwide epidemic spread and the impact of air traffic on it, we cannot simulate the 7 billions people. But we can represent only cities with airports and airplanes as agents. In this case, cities are entities with a population of millions inhabitants, that will not been spatially located. As we are only interested in the disease spread, we are only interested in the number of infected people in the cities (and susceptibles and recovered too). As a consequence, it appears particularly relevant to describe the evolution of the disease in the city using a ODE system.

In addition these models have the advantage to not be sensible to population size in the integration process. Dozens or billions people does not bring a computation time increase, contrarily to agent-based models.

Use of ODE in a GAML model

A stereotypical use of ODE in a GAMA agent-based model is to describe species where some agents attributes evolution is described using an ODE system.

As a consequence, the GAML language has been increased by two main concepts (as two statements): * equations can be written with the `equation` statement. An `equation` block is composed of a set of diff statement describing the evolution of species attributes. * an equation can be numerically integrated using the solve statement

equation

Defining an ODE system

Defining a new ODE system needs to define a new equation block in a species. As example, the following eqSI system describes the evolution of a population with 2 compartments (S and I) and the flow from S to I compartment:

```
species userSI {
    float t ;
    float I ;
    float S ;
    int N ;
    float beta<-0.4 ;
    float h ;

    equation eqSI {
        diff(S,t) = -beta * S * I / N ;
        diff(I,t) = beta * S * I / N ;
    }
}
```

This equation has to be defined in a species with t, S and I attributes. beta (and other similar parameters) can be defined either in the specific species (if it is specific to each agents) or in the global if it is a constant.

Note: the t attribute will be automatically updated using the solve statement ; it contains the time elapsed in the equation integration.

Using a built-in ODE system

In order to ease the use of very classical ODE system, some built-in systems have been implemented in GAMA. For example, the previous SI system can be written as follows. Three additional facets are used to define the system: * type: the identifier of the built-in system (here SI) (the list of all built-in systems are described below), * vars: this facet is expecting a list of variables of the species, that will be matched with the variables of the system, * `params`: this facet is expecting a list of variables of the species (of of the global), that will be matched with the parameters of the system.

```
equation eqBuiltInSI type: SI vars: [S,I,t] params: [N,beta] ;
```

Split a system into several agents

An equation system can be split into several species and each part of the system are synchronized using the `simultaneously` facet of `equation`. The system split into several agents can be integrated using a single call to the `solve` statement. Notice that all the `equation` definition must have the same name.

For example the SI system presented above can be defined in two different species `S_agt` (containing the equation defining the evolution of the S value) and `I_agt` (containing the equation defining the evolution of the I value). These two equations are linked using the `simultaneously` facet of the `equation` statement. This facet expects a set of agents. The integration is called only once in a simulation step, e.g. in the `S_agt` agent.

```
species S_agt {
    float t ;
    float Ssize ;

    equation evol simultaneously: [ I_agt ] {
        diff(Ssize, t) = (- sum(I_agt accumulate [each.beta *
   each.Isize]) * self.Ssize / N);
    }

    reflex solving {solve evol method : rk4 step : hKR4 ;}
}

species I_agt {
    float t ;
    float Isize ; // number of infected
    float beta ;

    equation evol simultaneously : [ S_agt ] {
        diff(Isize, t) = (beta * first(S_agt).Ssize * Isize / N);
    }
}
```

The interest is that the modeler can create several agents for each compartment, which different values. For example in the SI model, the modeler can choose to create 1 agent `S_agt` and 2 agents `I_agt`. The `beta` attribute will have different values in the two agents, in order to represent 2 different strains.

```
global {
    int number_S <- 495 ; // The number of susceptible
    int number_I <- 5   ; // The number of infected
    int nb_I <- 2;
    float gbeta  <- 0.3  ; // The parameter Beta

    int N <- number_S + nb_I * number_I ;
    float hKR4 <- 0.1 ;

    init {
        create S_agt {
            Ssize <- float(number_S) ;
        }
        create I_agt number: nb_I {
            Isize <- float(number_I) ;
            self.beta <- myself.gbeta + rnd(0.5) ;
        }
    }
}
```

The results are computed using the RK4 method with: * number_S = 495 * number_I = 5 * nb_I = 2 * gbeta = 0.3 * hKR4 = 0.1

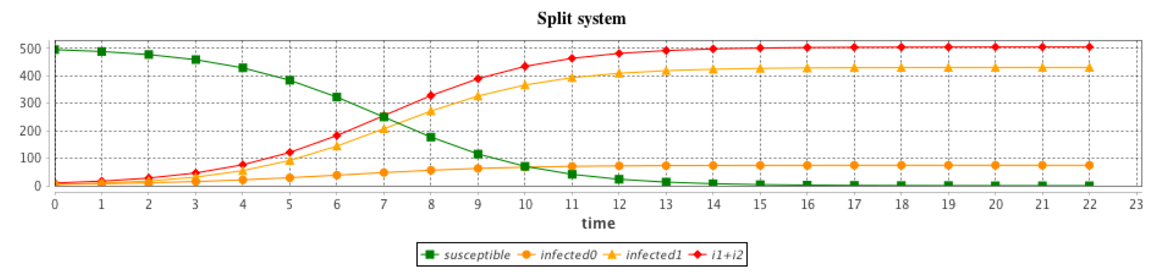


Figure 46.4: SI-split-results.png

solve an equation

The solve statement has been added in order to integrate numerically the equation system. It should be add into a reflex. At each simulation step, a step of the integration is executed, the length of the integration step is defined in the step facet. The solve statement will update the variables used in the equation system. The chosen integration method (defined in method) is Runge-Kutta 4 (which is very often a good choice of integration method in terms of accuracy).

```
reflex solving {
    solve eqSI method:rk4 step:h;
}
```

With a smaller integration step, the integration will be faster but less accurate.

More details

Details about the solve statement

The solve statement can have a huge set of facets (see [S_Statements#solve] for more details). The basic use of the solve statement requiers only the equation identifier. By default, the integration method is Runge-Kutta 4 with an integration step of 1, which means that at each simulation step the equation integration is made over 1 unit of time (which is implicitly defined by the system parameter value).

```
solve eqSI ;
```

2 integration methods can be used: * method: rk4 will use the Runge-Kutta 4 integration method * method: dp853 will use the Dorman-Prince 8(5,3) integration method. The advantage of this method compared to Runge-Kutta is that it has an evaluation of the error and can use it to adapt the integration step size.

In order to synchronize the simulation step and the equation integration step, 2 facets can be used: * step: number * cycle_length: number

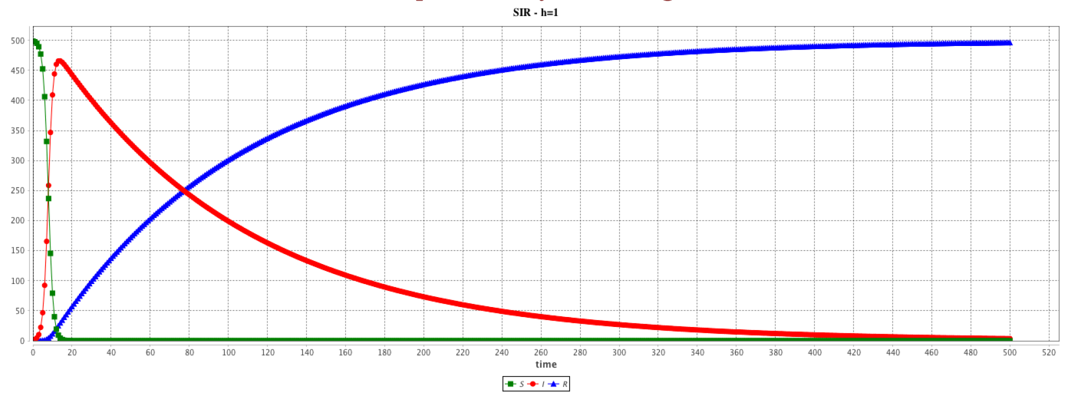
cycle_length (int): length of simulation cycle which will be synchronize with step of integrator (default value: 1) step (float): integration step, use with most integrator methods (default value: 1) time_final (float): target time for the integration (can be set to a value smaller than t0 for backward integration) time_initial (float): initial time discretizing_step (int): number of discret beside 2 step of simulation (default value: 0) integrated_times (list): time interval inside integration process integrated_values (list): list of variables's value inside integration process

Some facets are specific to the DP853 integration methods: `max_step`, `min_step`, `scalAbsoluteTolerance` and `scalRelativeTolerance`.

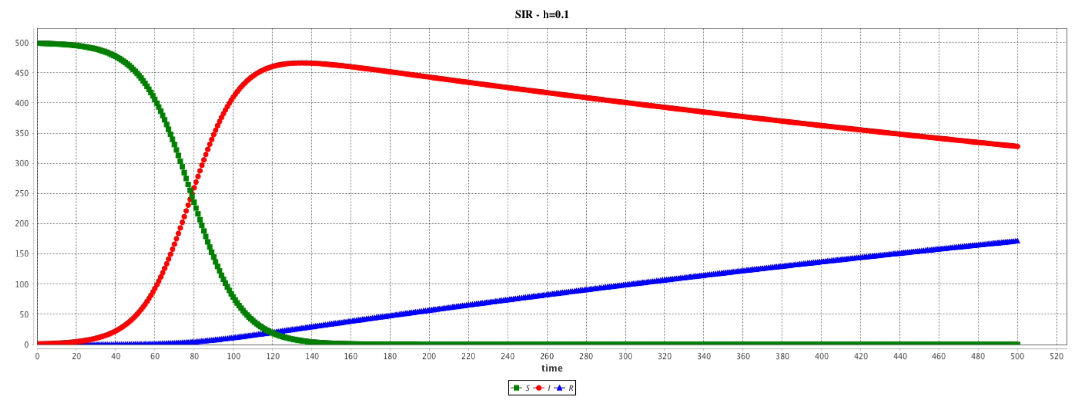
Example of the influence of the integration step

The `step` and `cycle_length` facets of the integration method may have a huge influence on the results. `step` has an impact on the result accuracy. In addition, it is possible to synchronize the step of the (agent-based) simulation and the (equation) integration step in various ways (depending on the modeler purpose) using the `cycle_length` facet: e.g. `cycle_length: 10` means that 10 simulation steps are equivalent to 1 unit of time of the integration method.

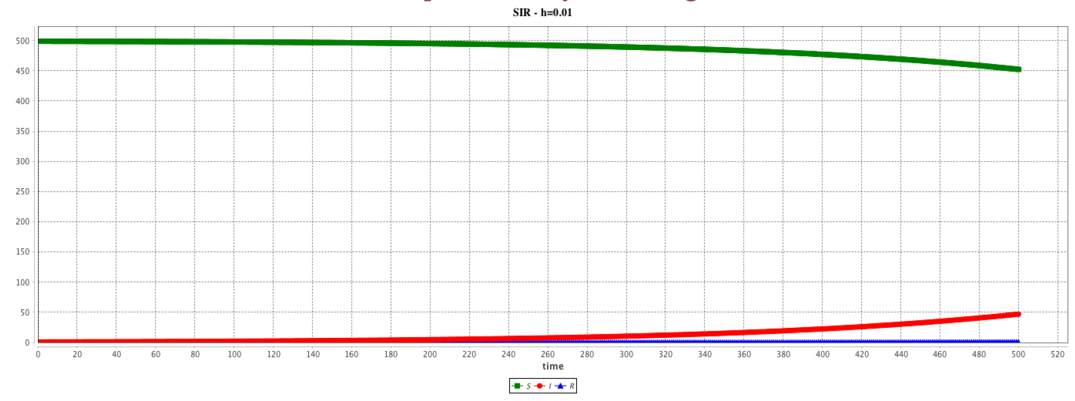
- `solve SIR method: "rk4" step: 1.0 cycle_length: 1.0 ;`



- `solve SIR method: "rk4" step: 0.1 cycle_length: 10.0 ;`



- solve SIR method: "rk4" step: 0.01 cycle_length: 100.0 ;



List of built-in ODE systems

Several built-in equations have been defined. `#### equation eqBuiltInSI type: SI vars: [S,I,t] params: [N,beta];`

This system is equivalent to:

```
equation eqSI {
    diff(S,t) = -beta * S * I / N ;
    diff(I,t) = beta * S * I / N ;
}
```

The results are provided using the Runge-Kutta 4 method using following initial values: * S = 499 * I = 1 * beta = 0.4 * h = 0.1

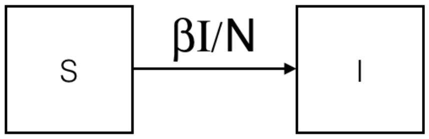


Figure 46.5: SI-compartment.png

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} \end{cases}$$

Figure 46.6: SI-equations.png

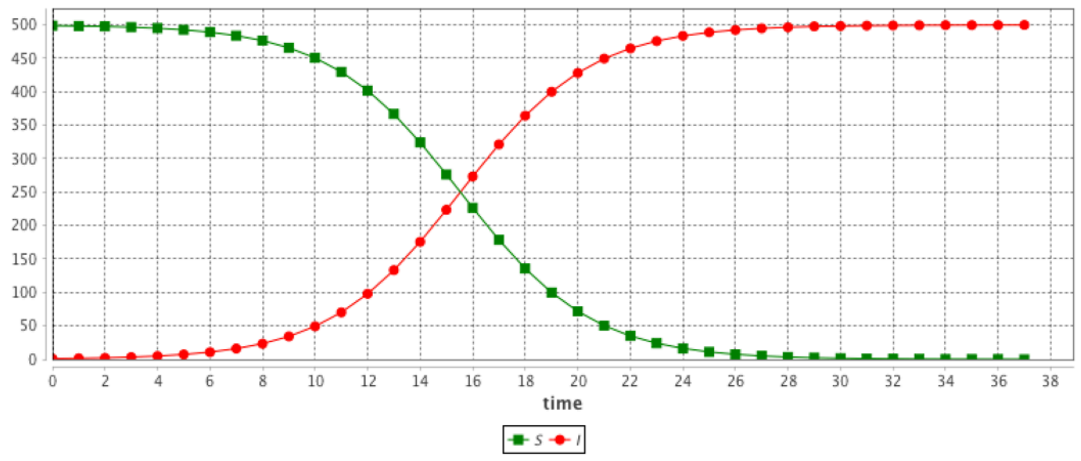


Figure 46.7: SI-result.png

`equation eqSIS type: SIS vars: [S,I,t] params: [N,beta,gamma];`

This system is equivalent to:

```
equation eqSIS {
    diff(S,t) = -beta * S * I / N + gamma * I;
    diff(I,t) = beta * S * I / N - gamma * I;
}
```

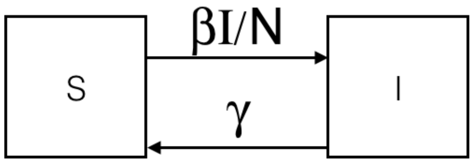
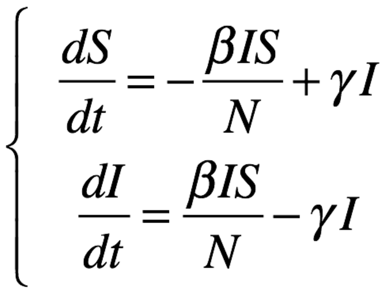


Figure 46.8: SIS-compartment.png



$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} + \gamma I \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I \end{cases}$$

Figure 46.9: SIS-equations.png

The results are provided using the Runge-Kutta 4 method using following initial values: * S = 499 * I = 1 * beta = 0.4 * gamma = 0.1 * h = 0.1

`equation eqSIR type:SIR vars:[S,I,R,t] params:[N,beta,gamma] ;`

This system is equivalent to:

```
equation eqSIR {
    diff(S,t) = (- beta * S * I / N);
    diff(I,t) = (beta * S * I / N) - (gamma * I);
```

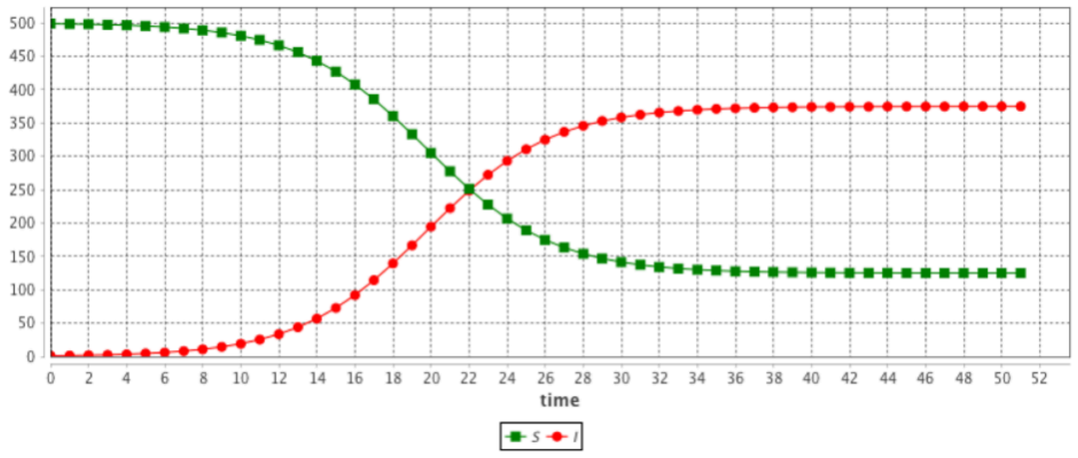


Figure 46.10: SIS-result.png

```
    diff(R,t) = (gamma * I);
}
```

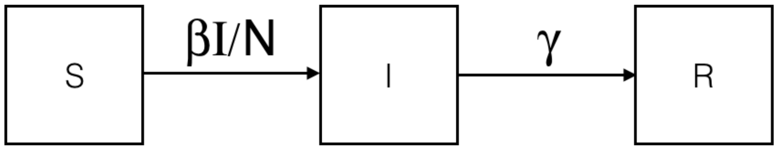


Figure 46.11: SIR-compartment.png

The results are provided using the Runge-Kutta 4 method using following initial values: * S = 499 * I = 1 * R = 0 * beta = 0.4 * gamma = 0.1 * h = 0.1

`equation eqSIRS type: SIRS vars: [S,I,R,t] params: [N,beta,gamma,omega,mu] ;`

This system is equivalent to:

```
equation eqSIRS {
     diff(S,t) = mu * N + omega * R + - beta * S * I / N - mu * S
    ;
     diff(I,t) = beta * S * I / N - gamma * I - mu * I ;
```

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I \\ \dfrac{dR}{dt} = \gamma I \end{cases}$$

Figure 46.12: SIR-equations.png

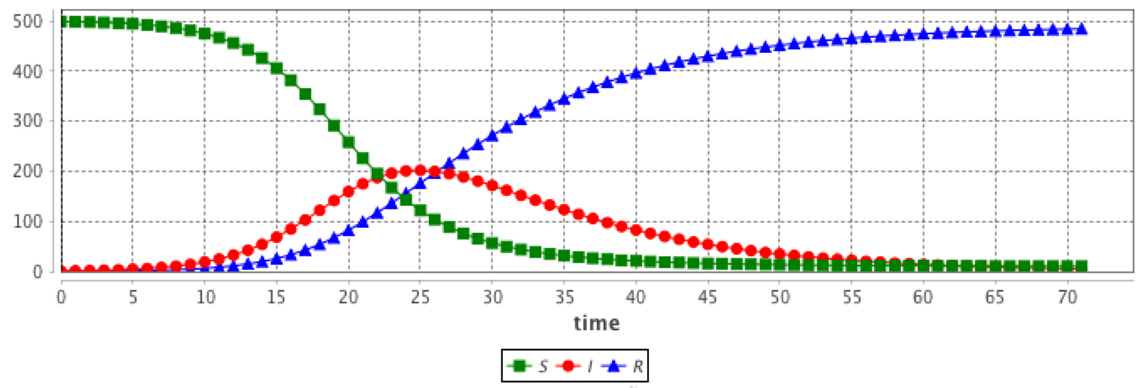


Figure 46.13: SIR-result.png

```
    diff(R,t) = gamma * I - omega * R - mu * R ;
}
```

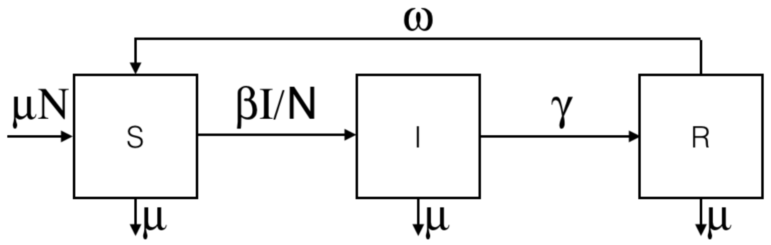


Figure 46.14: SIRS-compartment.png

$$\begin{cases} \dfrac{dS}{dt} = \mu N + \omega R - \dfrac{\beta IS}{N} - \mu S \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I - \mu I \\ \dfrac{dR}{dt} = \gamma I - \omega R - \mu R \end{cases}$$

Figure 46.15: SIRS-equations.png

The results are provided using the Runge-Kutta 4 method using following initial values: * S = 499 * I = 1 * R = 0 * beta = 0.4 * gamma = 0.01 * omega = 0.05 * mu = 0.01 * h = 0.1

`equation eqSEIR type: SEIR vars: [S,E,I,R,t] params: [N,beta,gamma,sigma,mu] ;`

This system is equivalent to:

```
equation eqSEIR {
    diff(S,t) = mu * N - beta * S * I / N - mu * S ;
    diff(E,t) = beta * S * I / N - mu * E - sigma * E ;
```

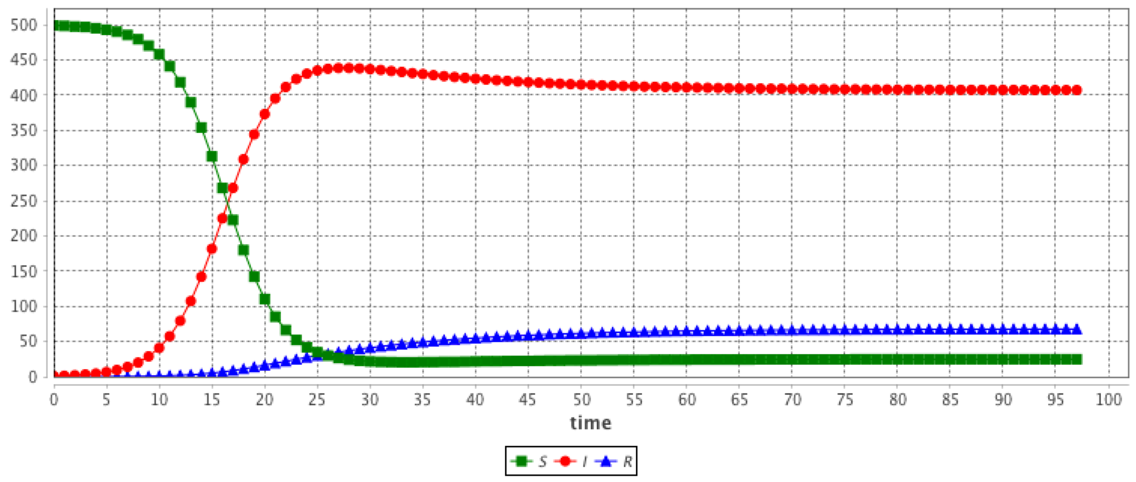


Figure 46.16: SIRS-result.png

```
    diff(I,t) = sigma * E - mu * I - gamma * I;
    diff(R,t) = gamma * I - mu * R ;
}
```

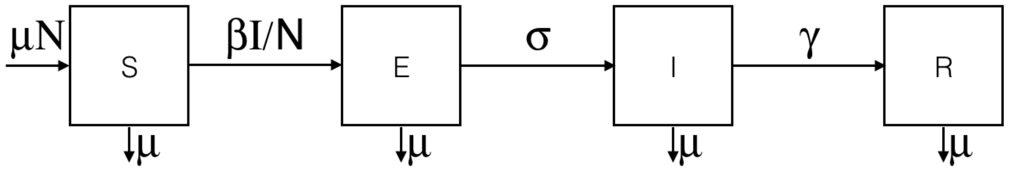


Figure 46.17: SEIR-compartment.png

The results are provided using the Runge-Kutta 4 method using following initial values: * S = 499 * E = 0 * I = 1 * R = 0 * beta = 0.4 * gamma = 0.01 * sigma = 0.05 * mu = 0.01 * h = 0.1

equation eqLV type: LV vars: [x,y,t] params: [alpha,beta,delta,gamma] ;

This system is equivalent to:

```
equation eqLV {
    diff(x,t) =   x * (alpha - beta * y);
```

$$\begin{cases} \dfrac{dS}{dt} = \mu N - \dfrac{\beta IS}{N} - \mu S \\ \dfrac{dE}{dt} = \dfrac{\beta IS}{N} - \sigma E - \mu E \\ \dfrac{dI}{dt} = \sigma E - \gamma I - \mu I \\ \dfrac{dR}{dt} = \gamma I - \mu R \end{cases}$$

Figure 46.18: SEIR-equations.png

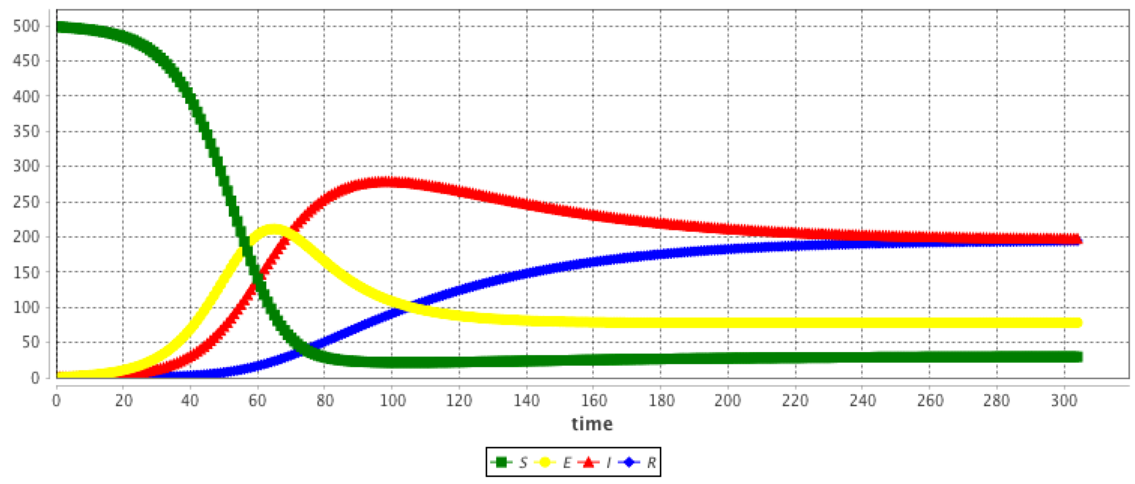


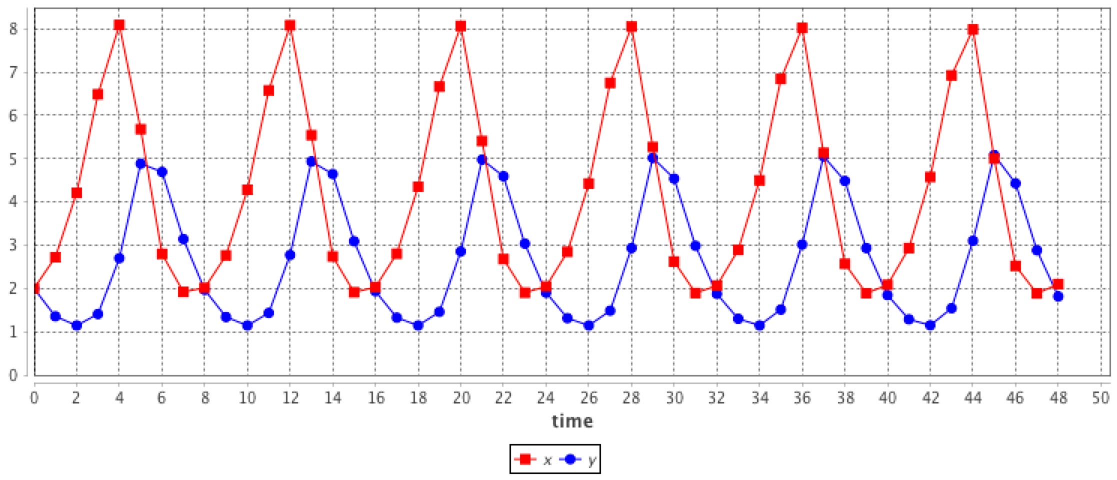
Figure 46.19: SEIR-result.png

```
    diff(y,t) = - y * (delta - gamma * x);
}
```

$$\begin{cases} \dfrac{dx}{dt} = x*(\alpha - \beta y) \\ \dfrac{dy}{dt} = -y*(\delta - \gamma x) \end{cases}$$

Figure 46.20: LV-equations.png

The results are provided using the Runge-Kutta 4 method using following initial values: * x = 2 * y = 2 * alpha = 0.8 * beta = 0.3 * gamma = 0.2 * delta = 0.85 * h = 0.1



//: # (endConcept|equation)

Recipes

Understanding the structure of models in GAML and gaining some insight of the language is required, but is usually not sufficient to build correct models or models that need to deal with specific approaches (like equation-based modeling). This section is intended to provide readers with practical "how to"s on various subjects, ranging from the use of database access to the design of agent communication languages. It is by no means exhaustive, and will progressively be extended with more "recipes" in the future, depending on the concrete questions asked by users.

Chapter 47

Manipulate OSM Datas

This section will be presented as a quick tutorial, showing how to proceed to manipulate OSM (Open street map) datas, and load them into GAMA. We will use the software QGIS to change the attributes of the OSM file.

From the website openstreetmap.org, we will chose a place (in this example, we will take a neighborhood in New York City). Directly from the website, you can export the chosen area in the osm format.

We have now to manipulate the attributes for the exported osm file. Several software are possible to use, but we will focus on QGIS, which is totally free and provides a lot of possibilities in term of manipulation of data.

Once you have installed correctly QGIS, launch QGIS Desktop, and start to import the topology from the osm file.

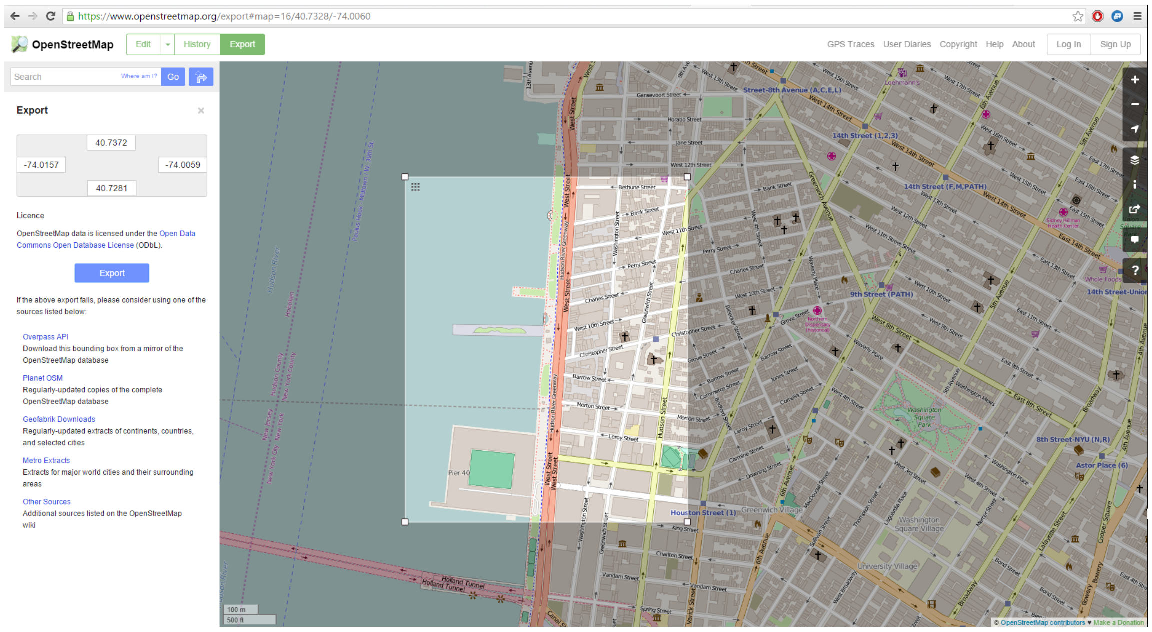
A message indicates that the import was successful. An output file .osm.db is created. You have now to export the topology to SpatiaLite.

Specify the path for your DataBase file, then choose the export type (in your case, we will choose the type "Polygons (closed ways)"), choose an output layer name. If you want to use the open street maps attributes values, click on "Load from DB", and select the attributes you want to keep. Click OK then.

A message indicates that the export was successful, and you have now a new layer created.

We will now manipulate the attributes of your datafile. Right click on the layer, and select "Open Attribute Table".

The table of attribute appears. Select the little pencil on the top-left corner of the window to modify the table.


https://www.openstreetmap.org/export#map=16/40.7328/-74.0060

Export

	40.7372	
-74.0157		-74.0059
	40.7281	

Licence

OpenStreetMap data is licensed under the Open Data Commons Open Database License (ODbL).

Export

If the above export fails, please consider using one of the sources listed below:

Overpass API
Download this bounding box from a mirror of the OpenStreetMap database

Planet OSM
Regularly-updated copies of the complete OpenStreetMap database

Geofabrik Downloads
Regularly-updated extracts of continents, countries, and selected cities

Metro Extracts
Extracts for major world cities and their surrounding areas

Other Sources
Additional sources listed on the OpenStreetMap wiki

Figure 47.1: images/manipulate_OSM_file_1.png

We will add an attribute manually. Click on the button "new column", choose a name and a type (we will choose the type "text").

A new column appears at the end of the table. Let's fill some values (for instance blue / red). Once you finishes, click on the "save edit" button.

Our file is now ready to be exported. Right click on the layer, and click on "Save As".

Choose "shapefile" as format, choose a save path and click ok.

Copy passed all the .shp created in the include folder of your GAMA project. You are now ready to write the model.

```
model HowToUseOpenStreetMap

global {
    // Global variables related to the Management units
    file shapeFile <- file('../includes/new_york.shp');

    //definition of the environment size from the shapefile.
    //Note that is possible to define it from several files by
   using: geometry shape <- envelope(envelope(file1) + envelope(
   file2) + ...);
```

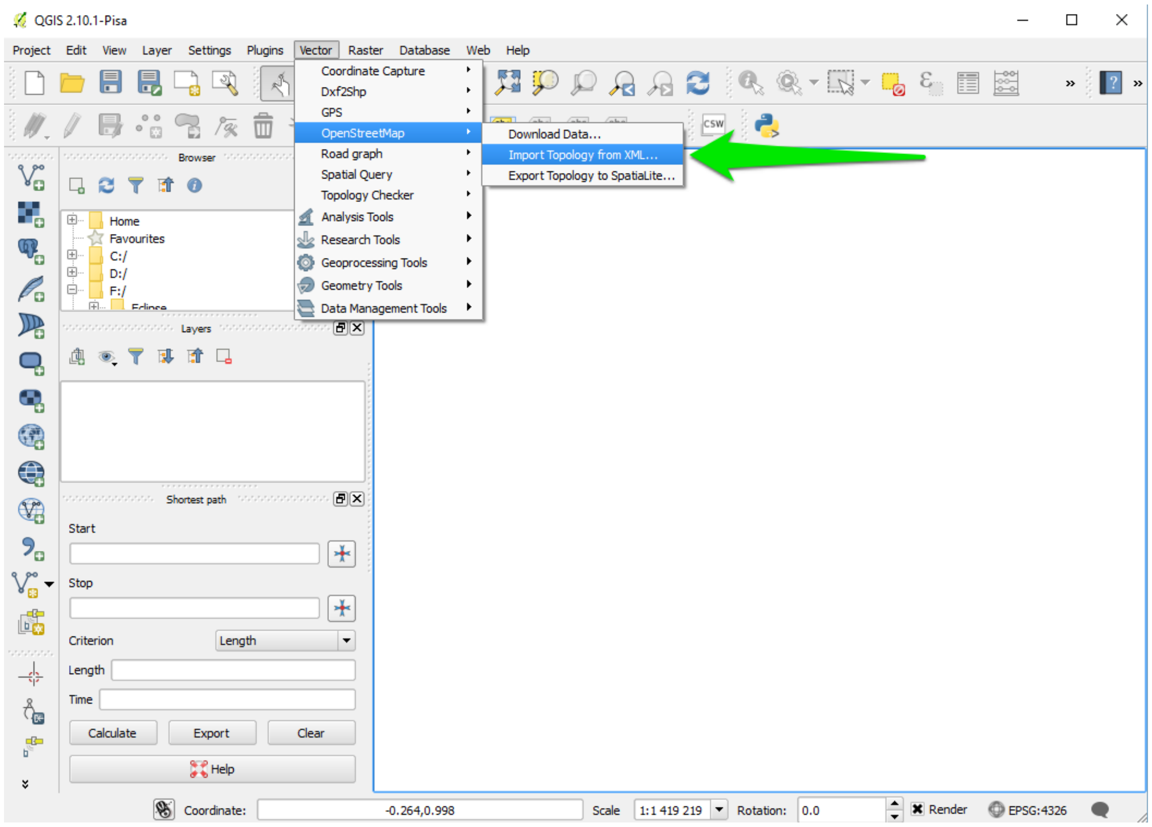


Figure 47.2: images/manipulate_OSM_file_2.png

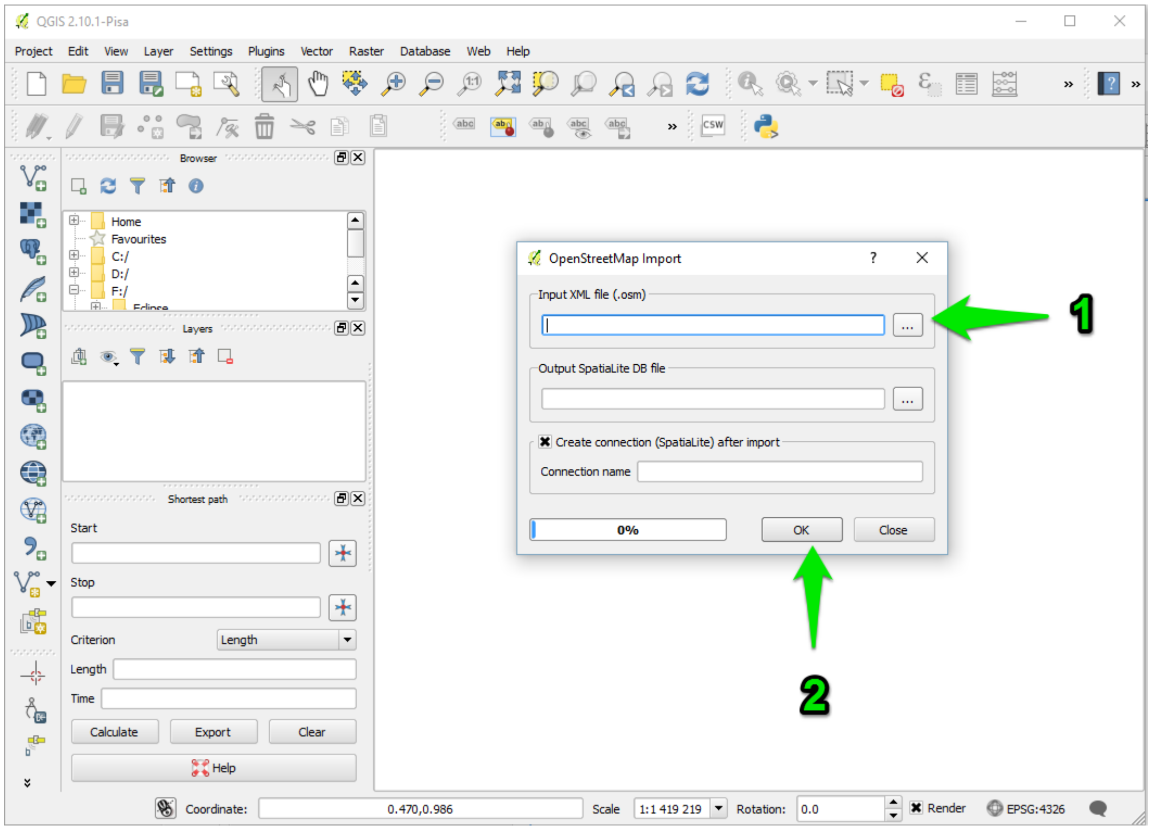


Figure 47.3: images/manipulate_OSM_file_3.png

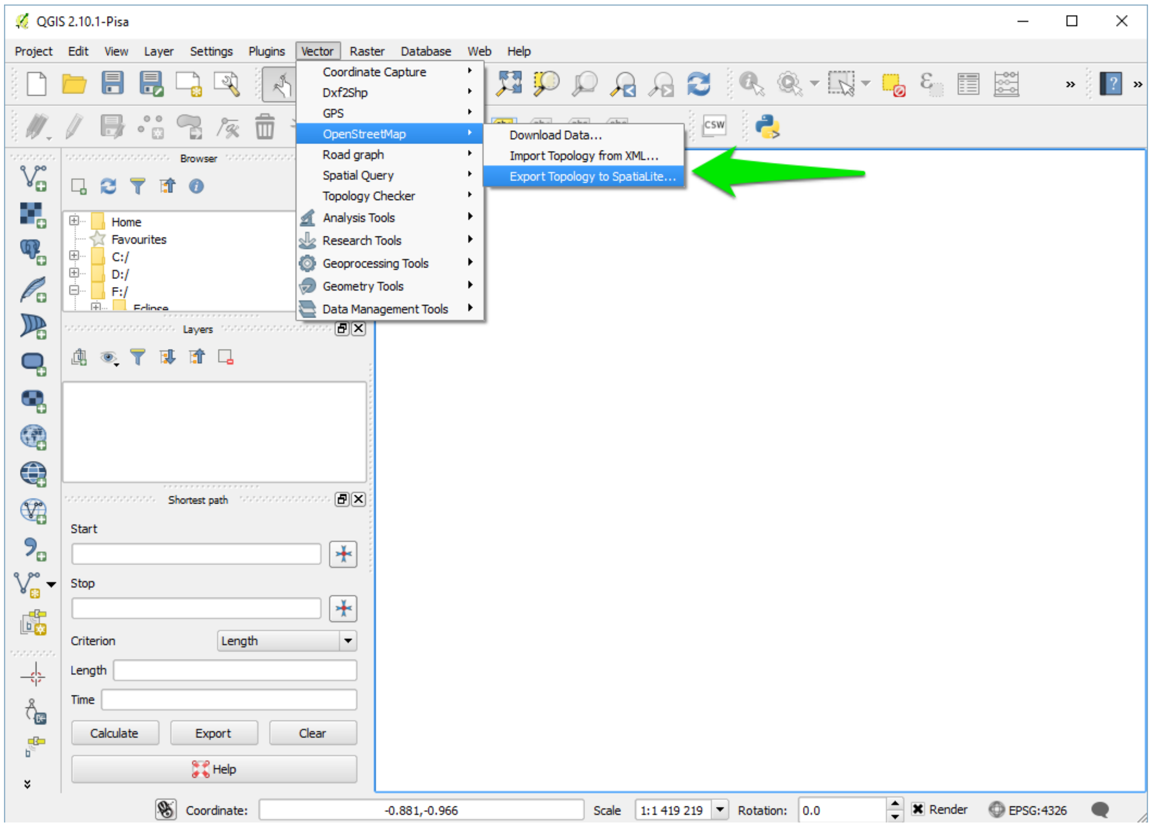
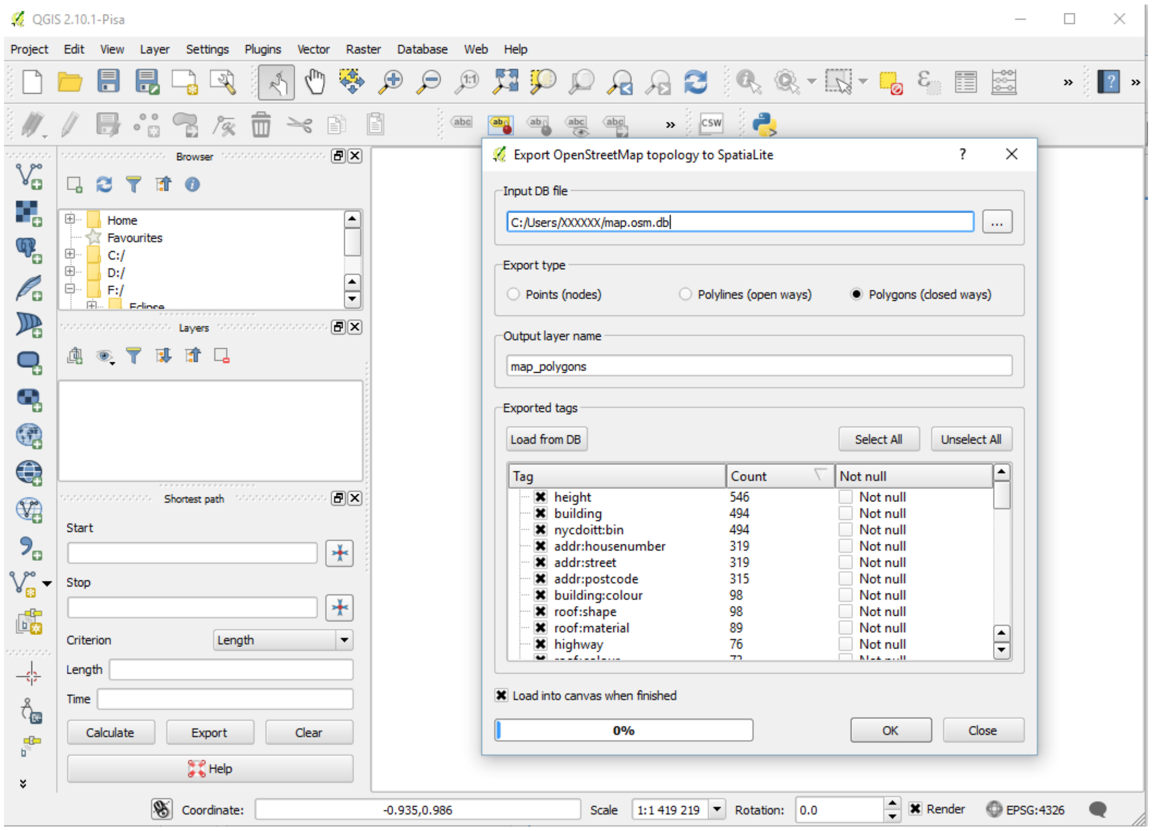


Figure 47.4: images/manipulate_OSM_file_4.png



Export OpenStreetMap topology to SpatiaLite

Input DB file: C:/Users/XXXXXX/map.osm.db

Export type: ○ Points (nodes) ○ Polylines (open ways) [● Polygons (closed ways)]

Output layer name: map_polygons

Exported tags

Load from DB | Select All | Unselect All

Tag	Count	Not null
☒ height	546	☐ Not null
☒ building	494	☐ Not null
☒ nycdoitt:bin	494	☐ Not null
☒ addr:housenumber	319	☐ Not null
☒ addr:street	319	☐ Not null
☒ addr:postcode	315	☐ Not null
☒ building:colour	98	☐ Not null
☒ roof:shape	98	☐ Not null
☒ roof:material	89	☐ Not null
☒ highway	76	☐ Not null

☒ Load into canvas when finished

0% | OK | Close

Start:
Stop:
Criterion: Length
Length:
Time:

Coordinate: -0.935,0.986 Scale 1:1 419 219 Rotation: 0.0 ☒ Render EPSG:4326

Figure 47.5: images/manipulate_OSM_file_5.png

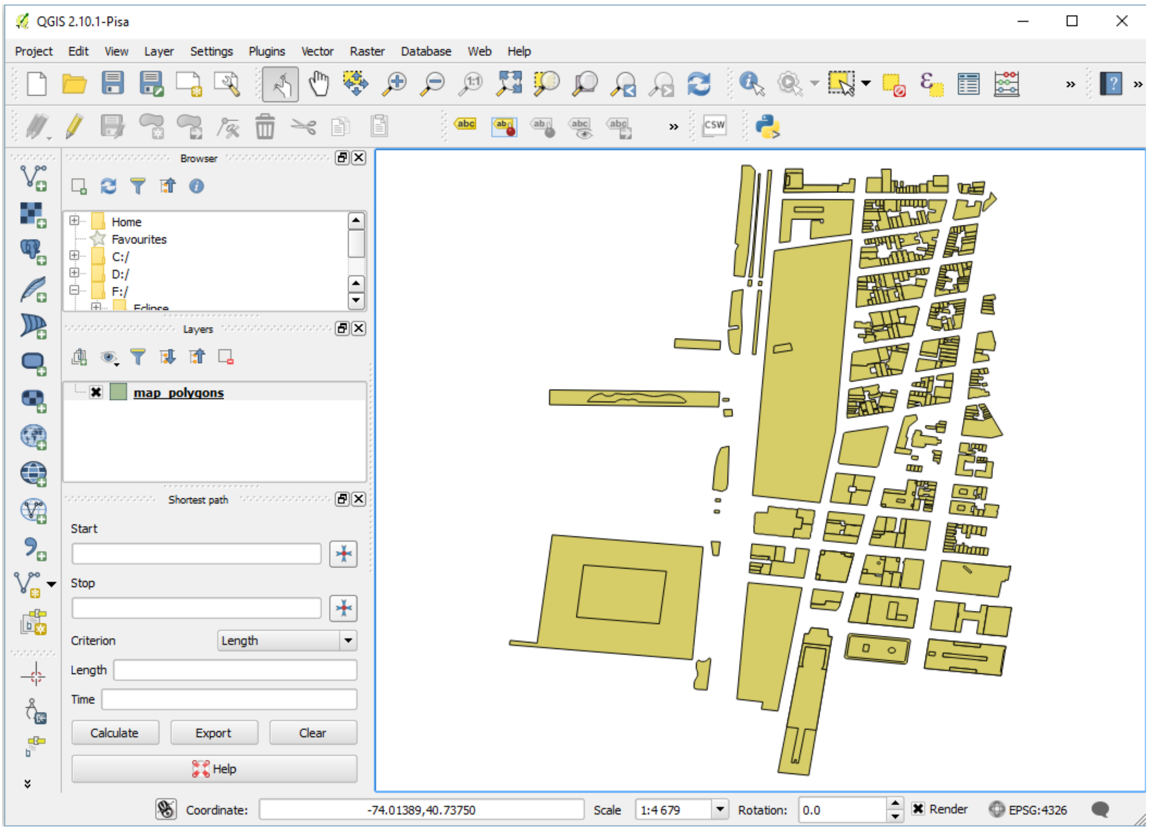


Figure 47.6: images/manipulate_OSM_file_6.png

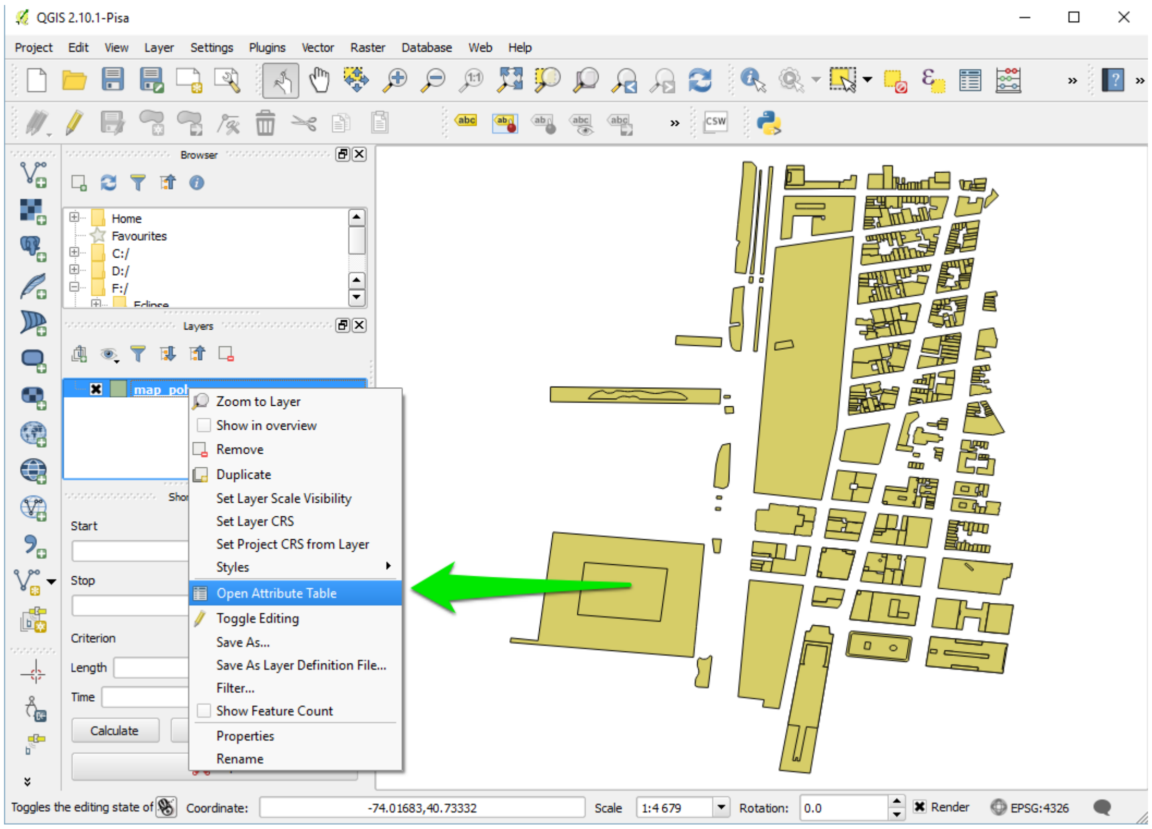
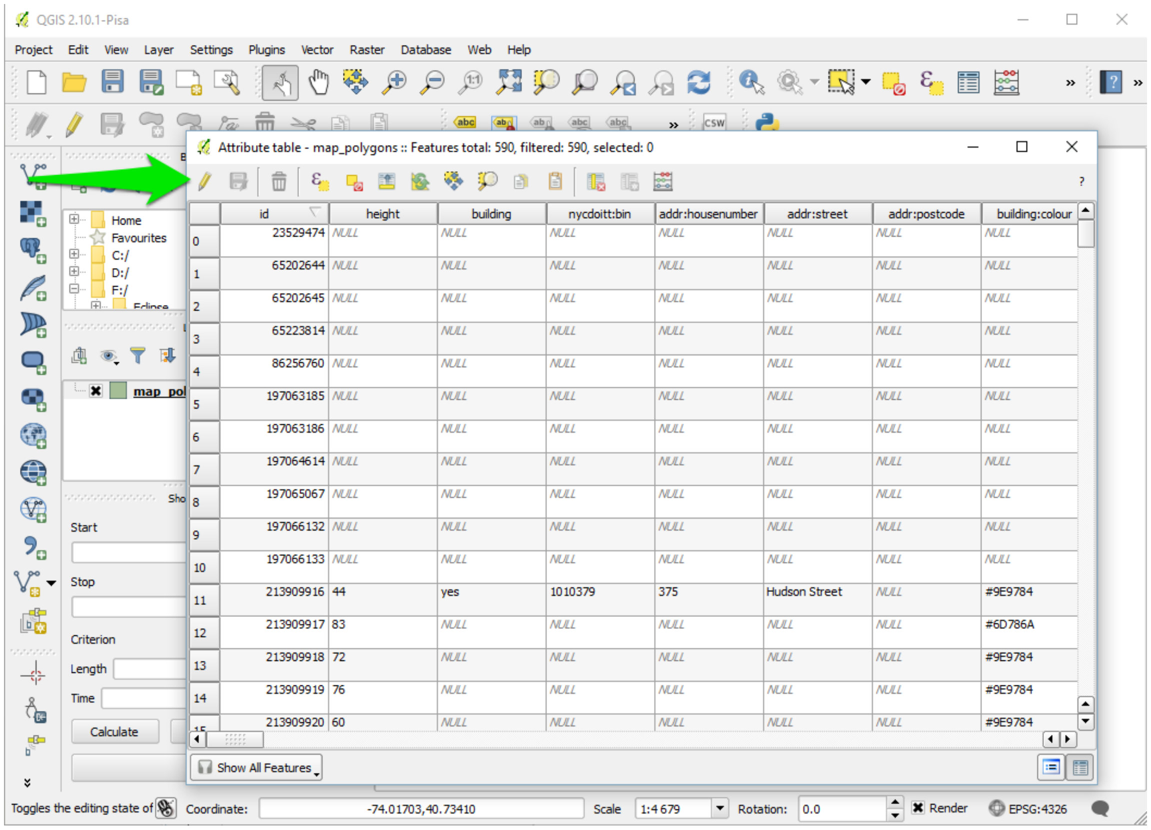


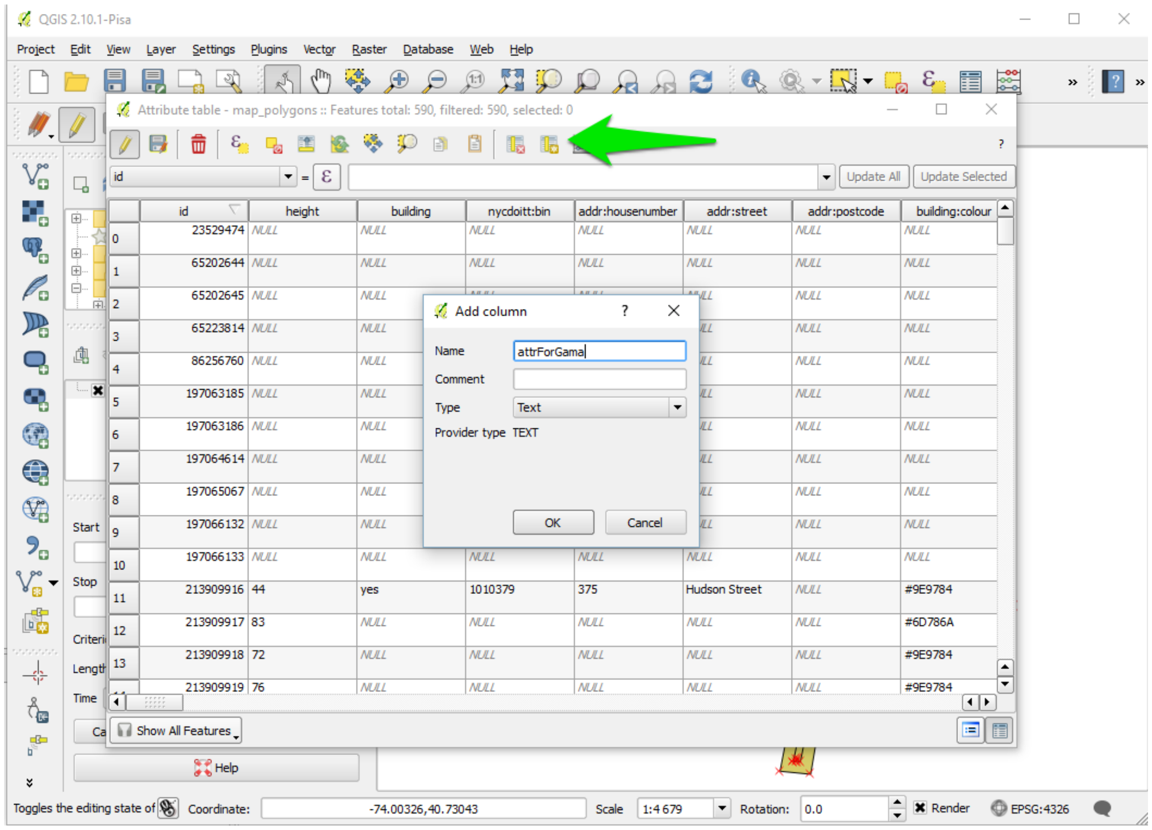
Figure 47.7: images/manipulate_OSM_file_7.png



Attribute table - map_polygons :: Features total: 590, filtered: 590, selected: 0

	id	height	building	nycdoitt:bin	addr:housenumber	addr:street	addr:postcode	building:colour
0	23529474	NULL	NULL	NULL	NULL	NULL	NULL	NULL
1	65202644	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	65202645	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	65223814	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	86256760	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	197063185	NULL	NULL	NULL	NULL	NULL	NULL	NULL
6	197063186	NULL	NULL	NULL	NULL	NULL	NULL	NULL
7	197064614	NULL	NULL	NULL	NULL	NULL	NULL	NULL
8	197065067	NULL	NULL	NULL	NULL	NULL	NULL	NULL
9	197066132	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10	197066133	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	213909916	44	yes	1010379	375	Hudson Street	NULL	#9E9784
12	213909917	83	NULL	NULL	NULL	NULL	NULL	#6D786A
13	213909918	72	NULL	NULL	NULL	NULL	NULL	#9E9784
14	213909919	76	NULL	NULL	NULL	NULL	NULL	#9E9784
15	213909920	60	NULL	NULL	NULL	NULL	NULL	#9E9784

Figure 47.8: images/manipulate_OSM_file_8.png



Attribute table - map_polygons :: Features total: 590, filtered: 590, selected: 0

id =

	id	height	building	nycdoitt:bin	addr:housenumber	addr:street	addr:postcode	building:colour
0	23529474	NULL	NULL	NULL	NULL	NULL	NULL	NULL
1	65202644	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	65202645	NULL	NULL			LL	NULL	NULL
3	65223814	NULL	NULL			LL	NULL	NULL
4	86256760	NULL	NULL			LL	NULL	NULL
5	197063185	NULL	NULL			LL	NULL	NULL
6	197063186	NULL	NULL			LL	NULL	NULL
7	197064614	NULL	NULL			LL	NULL	NULL
8	197065067	NULL	NULL			LL	NULL	NULL
9	197066132	NULL	NULL			LL	NULL	NULL
10	197066133	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	213909916	44	yes	1010379	375	Hudson Street	NULL	#9E9784
12	213909917	83	NULL	NULL	NULL	NULL	NULL	#6D786A
13	213909918	72	NULL	NULL	NULL	NULL	NULL	#9E9784
	213909919	76	NULL	NULL	NULL	NULL	NULL	#9E9784

Add column

Name: attrForGama

Comment:

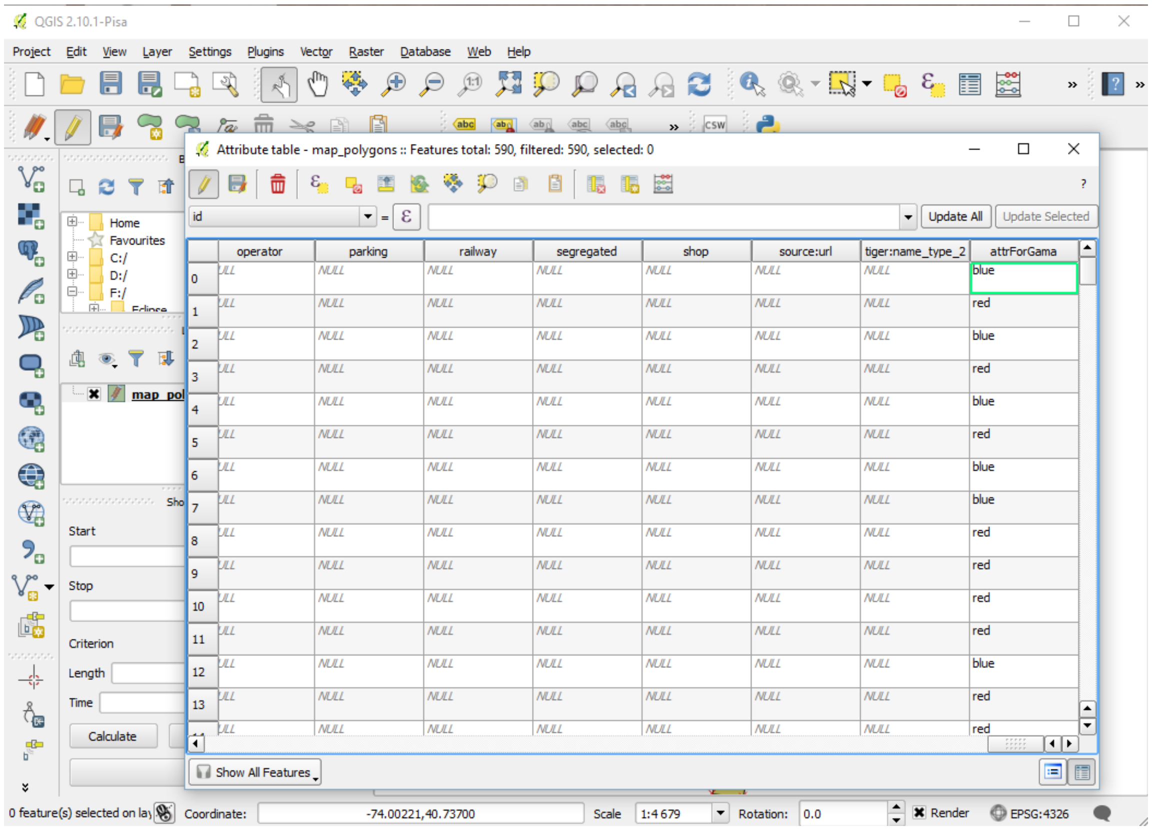
Type: Text

Provider type: TEXT

OK Cancel

Toggles the editing state of Coordinate: -74.00326,40.73043 Scale 1:4 679 Rotation: 0.0 Render EPSG:4326

Figure 47.9: images/manipulate_OSM_file_9.png



Attribute table - map_polygons :: Features total: 590, filtered: 590, selected: 0

	operator	parking	railway	segregated	shop	source:url	tiger:name_type_2	attrForGama
0	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
1	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
2	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
3	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
4	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
5	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
6	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
7	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
8	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
9	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
10	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
11	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
12	ULL	NULL	NULL	NULL	NULL	NULL	NULL	blue
13	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red
	ULL	NULL	NULL	NULL	NULL	NULL	NULL	red

0 feature(s) selected on lay Coordinate: -74.00221,40.73700 Scale 1:4 679 Rotation: 0.0 Render EPSG:4326

Figure 47.10: images/manipulate_OSM_file_10.png

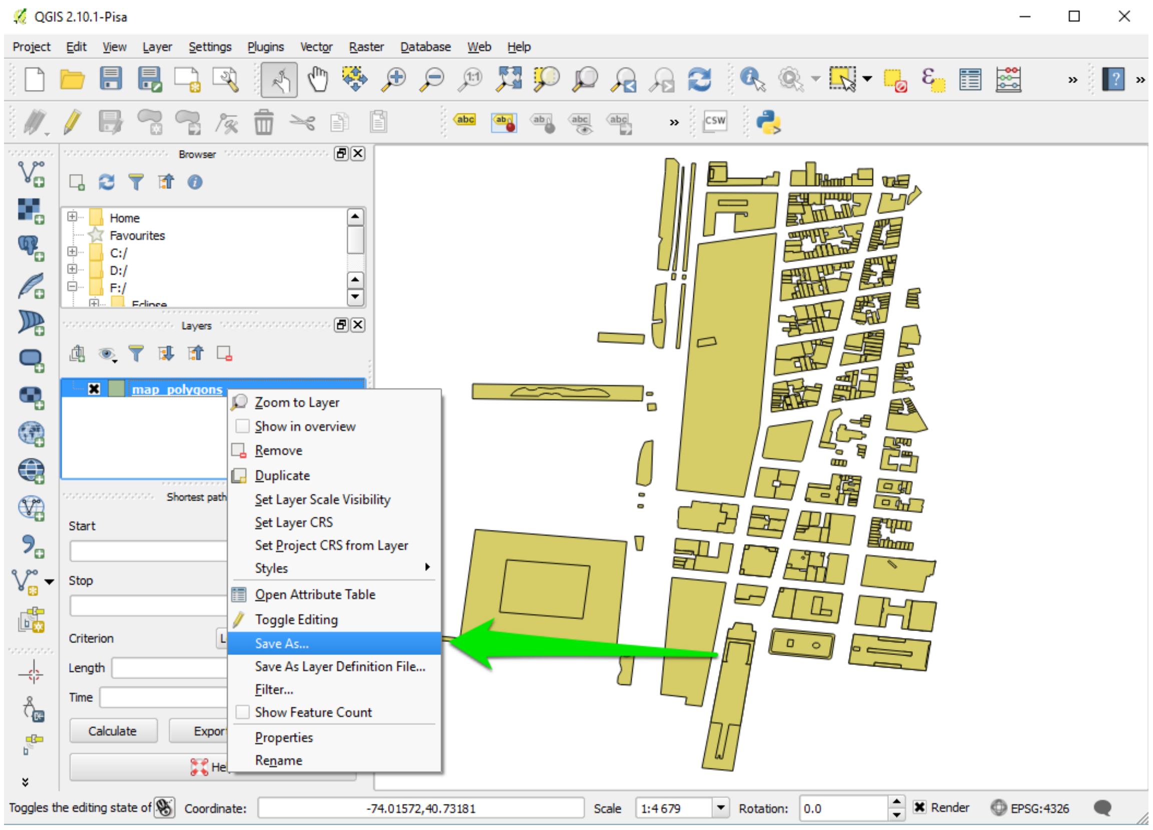
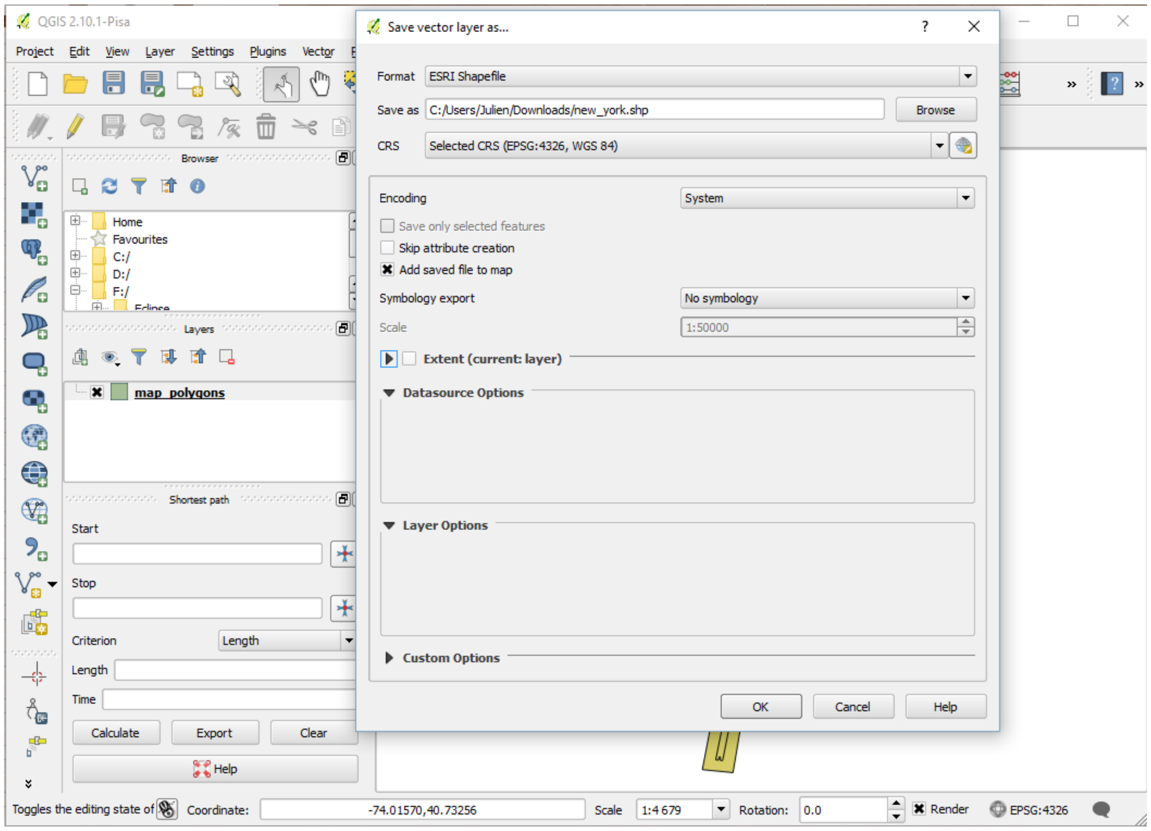


Figure 47.11: images/manipulate_OSM_file_11.png


Save vector layer as...

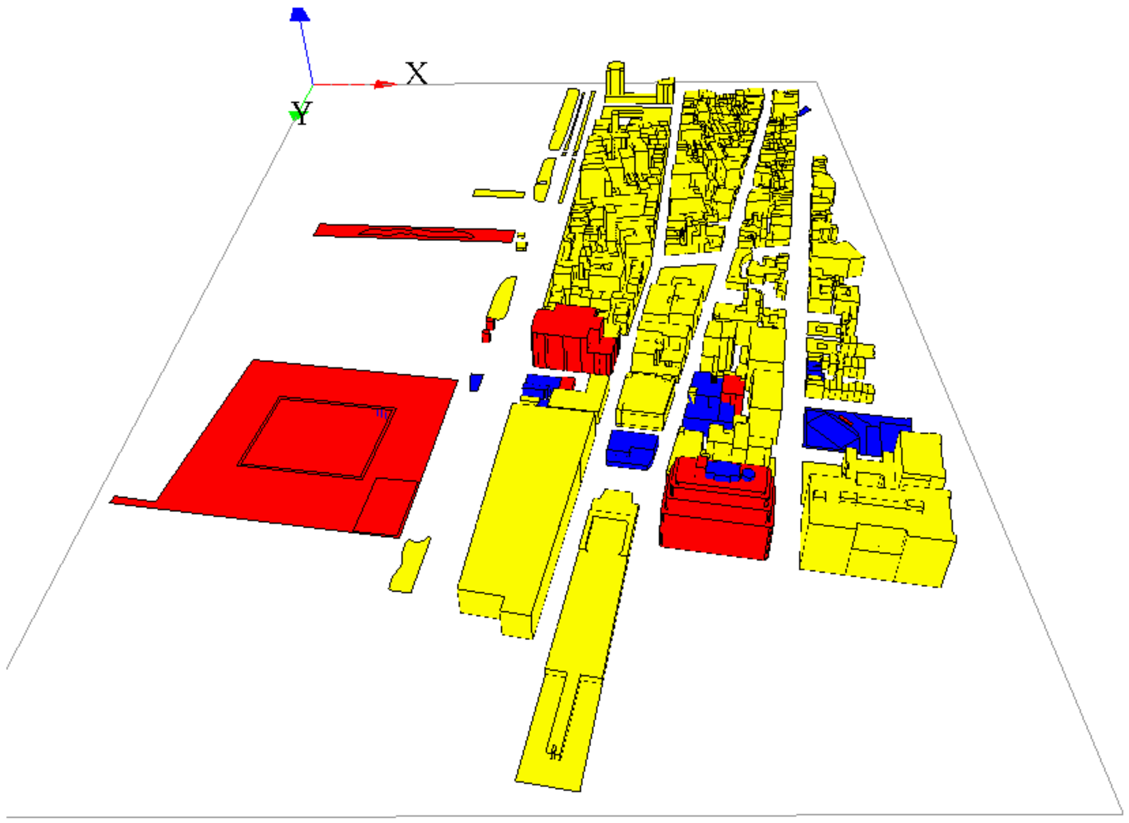
Format: ESRI Shapefile
Save as: C:/Users/Julien/Downloads/new_york.shp
CRS: Selected CRS (EPSG:4326, WGS 84)
Encoding: System
☐ Save only selected features
☐ Skip attribute creation
☒ Add saved file to map
Symbology export: No symbology
Scale: 1:50000
☐ Extent (current: layer)
Datasource Options
Layer Options
Custom Options

Coordinate: -74.01570,40.73256
Scale: 1:4 679
Rotation: 0.0
☒ Render
EPSG:4326

Figure 47.12: images/manipulate_OSM_file_12.png

```
    geometry shape <- envelope(shapeFile);

    init {
        //Creation of elementOfNewYork agents from the shapefile
   (and reading some of the shapefile attributes)
        create elementOfNewYork from: shapeFile
            with: [elementId::int(read('id')), elementHeight::int
   (read('height')), elementColor::string(read('attrForGama'))] ;
    }
}

species elementOfNewYork{
    int elementId;
    int elementHeight;
    string elementColor;

    aspect basic{
        draw shape color: (elementColor = "blue") ? #blue : ( (
   elementColor = "red") ? #red : #yellow ) depth: elementHeight;
    }
}

experiment main type: gui {
    output {
        display HowToUseOpenStreetMap type:opengl {
            species elementOfNewYork aspect: basic;
        }
    }
}
```

Here is the result, with a special colorization of the different elements regarding to the value of the attribute "attrForGama", and an elevation regarding to the value of the attribute "height".



//: # (endConcept|use_osm_datas)

Chapter 48

Implementing diffusion

GAMA provides you the possibility to represent and simulate the diffusion of a variable through a grid topology.

Index

Diffuse statement

The statement to use for the diffusion is `diffuse`. It has to be used in a `grid` species. The `diffuse` uses the following facets:

- `var` (an identifier), (omissible) : the variable to be diffused

- `on` (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the variation value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix masking the diffusion (matrix created from a image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- `matrix` (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- `min_value` (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- `propagation` (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbours of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : S' = (S / N / proportion) - variation. The intensity of S is then diminished by S * proportion on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals 'gradient', the original intensity is not modified, and each neighbors receives the intensity : S / proportion - variation. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to 'diffusion'.
- `proportion` (float): a diffusion rate
- `radius` (int): a diffusion radius (in number of cells from the center)
- `variation` (float): an absolute value to decrease at each neighbors

To write a diffusion, you first have to declare a grid, and declare a special attribute for the diffusion. You will then have to write the `diffuse` statement in an other scope (such as the `global` scope for instance), which will permit the values to be diffused at each step. There, you will specify which variable you want to diffuse (through the `var` facet), on which species or list of agents you want the diffusion (through the `on` facet), and how you want this value to be diffused (through all the other facets, we will see how it works with matrix and with special parameters just after).

Here is the template of code we will use for the next following part of this page:

```
global {
    int size <- 64; // the size has to be a power of 2.
    cells selected_cells;

    // Initialize the emiter cell as the cell at the center of
   the word
    init {
        selected_cells <- location as cells;
    }
    // Affecting "1" to each step
    reflex new_Value {
        ask(selected_cells){
            phero <- 1.0;
        }
    }

    reflex diff {
        // Declare a diffusion on the grid "cells" and on "
   quick_cells". The diffusion declared on "quick_cells" will
   make 10 computations at each step to accelerate the process.
        // The value of the diffusion will be store in the new
   variable "phero" of the cell.
        diffuse var: phero on: cells /*HERE WRITE DOWN THE
   DIFFUSION PROPERTIES*/;
    }
}


grid cells height: size width: size {
    // "phero" is the variable storing the value of the diffusion
    float phero  <- 0.0;
    // The color of the cell is linked to the value of "phero".
    rgb color <- hsb(phero,1.0,1.0) update: hsb(phero,1.0,1.0);
}


experiment diffusion type: gui {
    output {
        display a type: opengl {
            // Display the grid with elevation
            grid cells elevation: phero * 10 triangulation: true;
        }
    }
}
```

This model will simulate a diffusion through a grid at each step, affecting 1 to the center cell diffusing variable value. The diffusion will be seen during the simulation through a color code, and through the elevation of the cell.

Diffusion with matrix

A first way of specifying the behavior of your diffusion is using diffusion matrix. A diffusion matrix is a 2 dimension matrix `[n][m]` with `float` values, where both `n` and `m` have to be **pair values**. The most often, diffusion matrix are square matrix, but you can also declare rectangular matrix.

Example of matrix:

```
matrix<float> mat_diff <- matrix([
        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);
```

In the `diffuse` statement, you than have to specify the matrix of diffusion you want in the facet `matrix`.

```
diffuse var: phero on: cells matrix:mat_diff;
```

Using the facet `propagation`, you can specify if you want the value to be propagated as a *diffusion* or as a *gratient*.

Diffusion matrix

A *diffusion* (the default value of the facet `propagation`) will spread the values to the neighbors cells according to the diffusion matrix, and all those values will be added together, as it is the case in the following example :

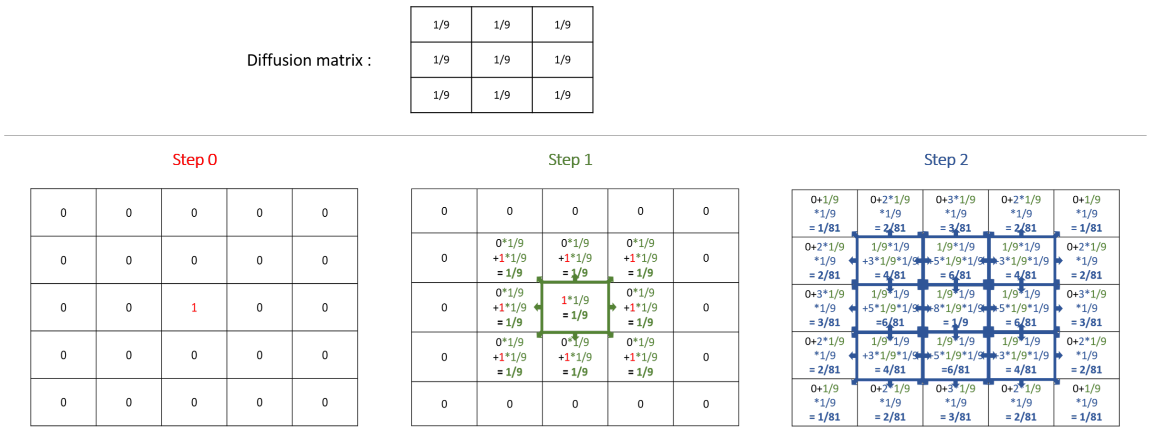


Figure 48.1: resources/images/recipes/diffusion_computation.png

Note that the sum of all the values diffused at the next step is equal to the sum of the values that will be diffused multiply by the sum of the values of the diffusion matrix. That means that if the sum of the values of your diffusion matrix is larger than 1, the values will increase exponentially at each step. The sum of the value of a diffusion matrix is usually equal to 1.

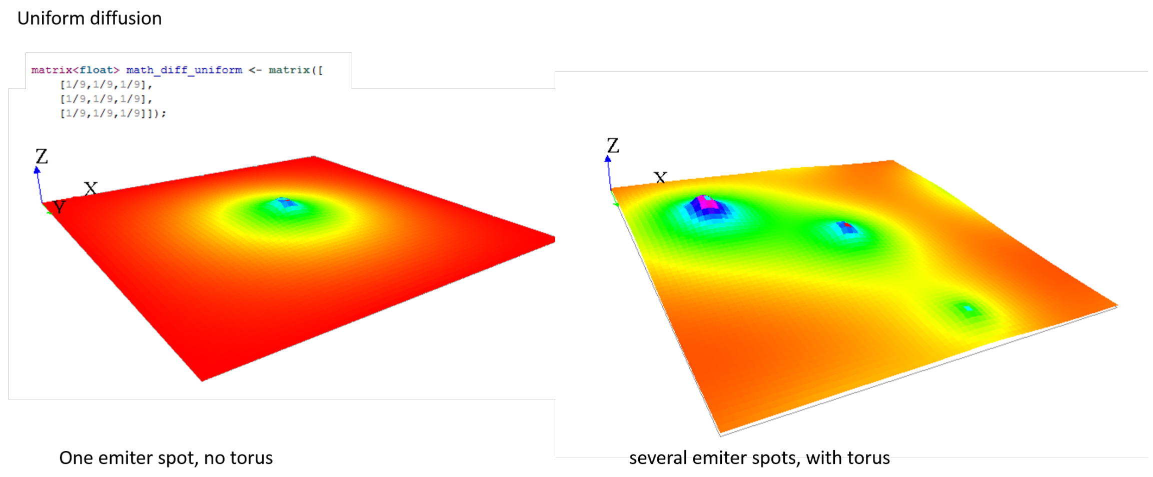
Here are some example of matrix you can use, played with the template model:

Gradient matrix

A `gradient` (use facet : `propagation:gradient`) is an other type of propagation. This time, only the larger value diffused will be chosen as the new one.

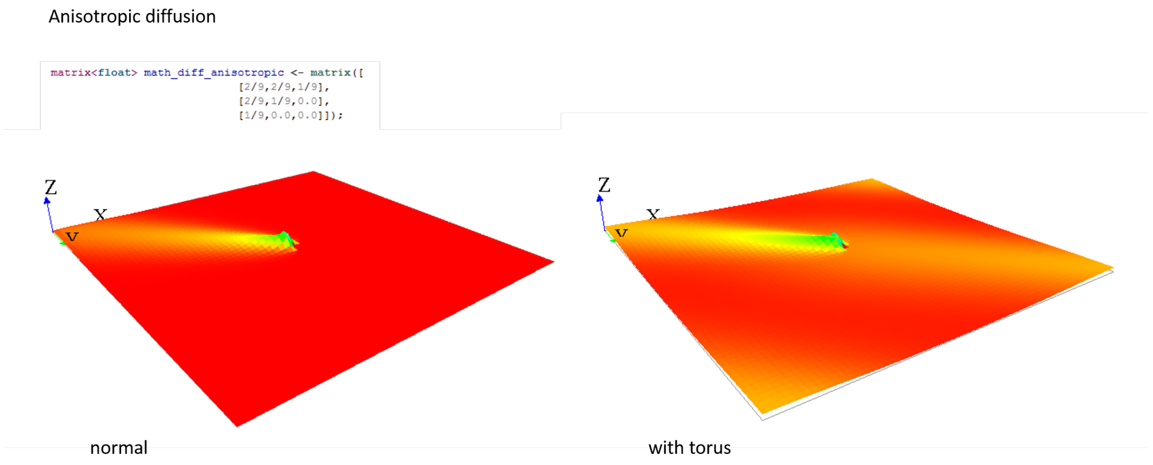
Note that unlike the *diffusion* propagation, the sum of your matrix can be greater than 1 (and it is the case, most often !).

Here are some example of matrix with gradient propagation:



```
matrix<float> math_diff_uniform <- matrix([
    [1/9,1/9,1/9],
    [1/9,1/9,1/9],
    [1/9,1/9,1/9]]);
```

Figure 48.2: resources/images/recipes/uniform_diffusion.png



```
matrix<float> math_diff_anisotropic <- matrix([
                                [2/9,2/9,1/9],
                                [2/9,1/9,0.0],
                                [1/9,0.0,0.0]]);
```

Figure 48.3: resources/images/recipes/anisotropic_diffusion.png

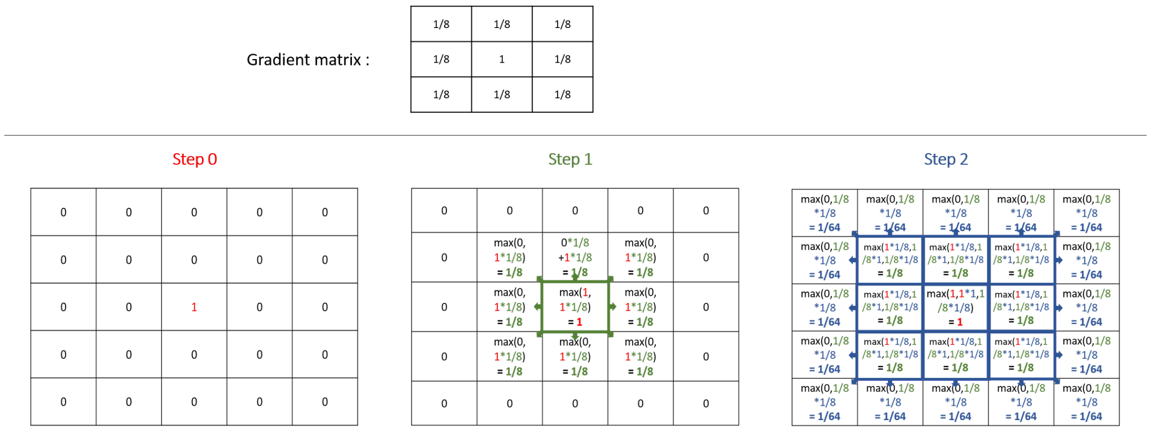
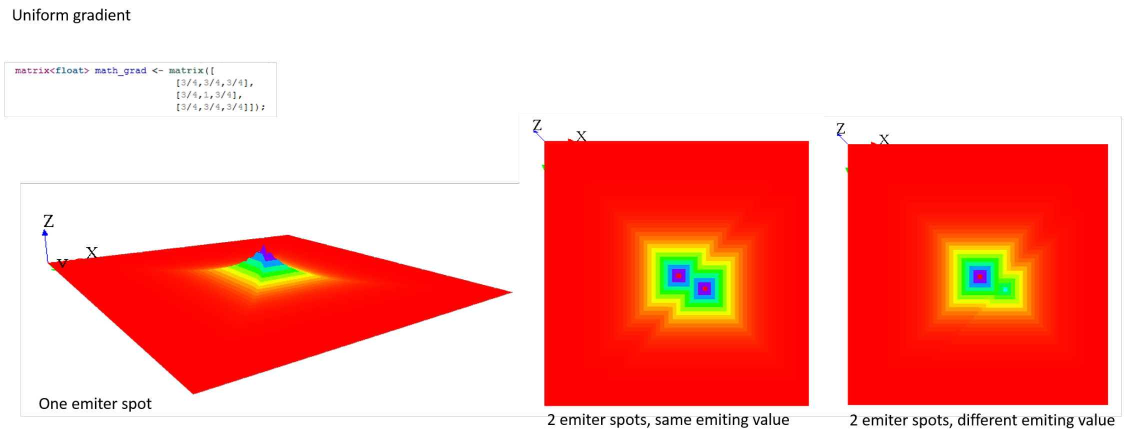
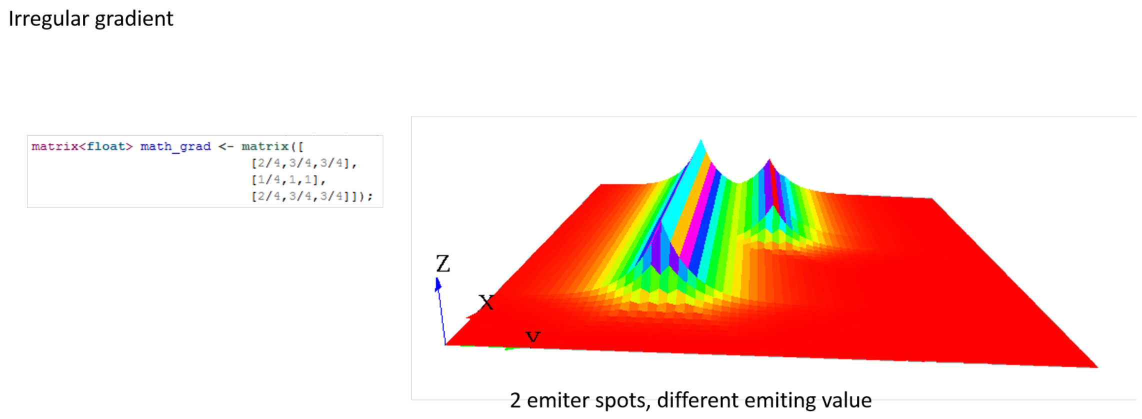


Figure 48.4: resources/images/recipes/gradient_computation.png


Uniform gradient
Z
X
Y
One emiter spot
2 emiter spots, same emiting value
2 emiter spots, different emiting value

```
matrix<float> math_grad <- matrix([
                    [3/4,3/4,3/4],
                    [3/4,1,3/4],
                    [3/4,3/4,3/4]]);
```

Figure 48.5: resources/images/recipes/uniform_gradient.png

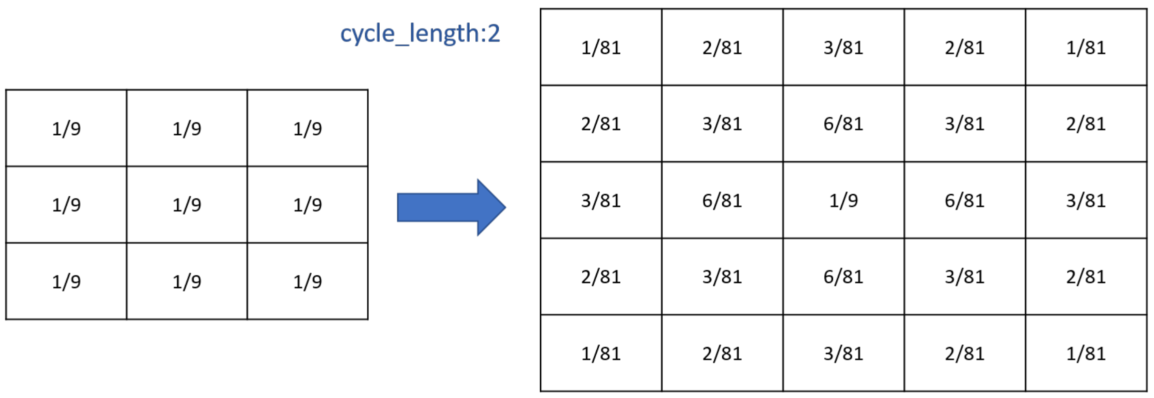


```
matrix<float> math_grad <- matrix([
                    [2/4,3/4,3/4],
                    [1/4,1,1],
                    [2/4,3/4,3/4]]);
```

Figure 48.6: resources/images/recipes/irregular_gradient.png

Compute multiple propagations at the same step

You can compute several times the propagation you want by using the facet `cycle_length`. GAMA will compute for you the corresponding new matrix, and will apply it.



1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

1/81	2/81	3/81	2/81	1/81
2/81	3/81	6/81	3/81	2/81
3/81	6/81	1/9	6/81	3/81
2/81	3/81	6/81	3/81	2/81
1/81	2/81	3/81	2/81	1/81

Figure 48.7: resources/images/recipes/cycle_length.png

Writing those two thinks are exactly equivalent (for diffusion):

```
    matrix<float> mat_diff <- matrix([
            [1/81,2/81,3/81,2/81,1/81],
            [2/81,4/81,6/81,4/81,2/81],
            [3/81,6/81,1/9,6/81,3/81],
            [2/81,4/81,6/81,4/81,2/81],
            [1/81,2/81,3/81,2/81,1/81]]);
    reflex diff {
        diffuse var: phero on: cells matrix:mat_diff;
```

```
    matrix<float> mat_diff <- matrix([
            [1/9,1/9,1/9],
            [1/9,1/9,1/9],
            [1/9,1/9,1/9]]);
    reflex diff {
        diffuse var: phero on: cells matrix:mat_diff cycle_length
    :2;
```

Executing several diffusion matrix

If you execute several times the statement `diffuse` with different matrix on the same variable, their values will be added (and centered if their dimension is not equal).

Thus, the following 3 matrix will be combined to create one unique matrix:

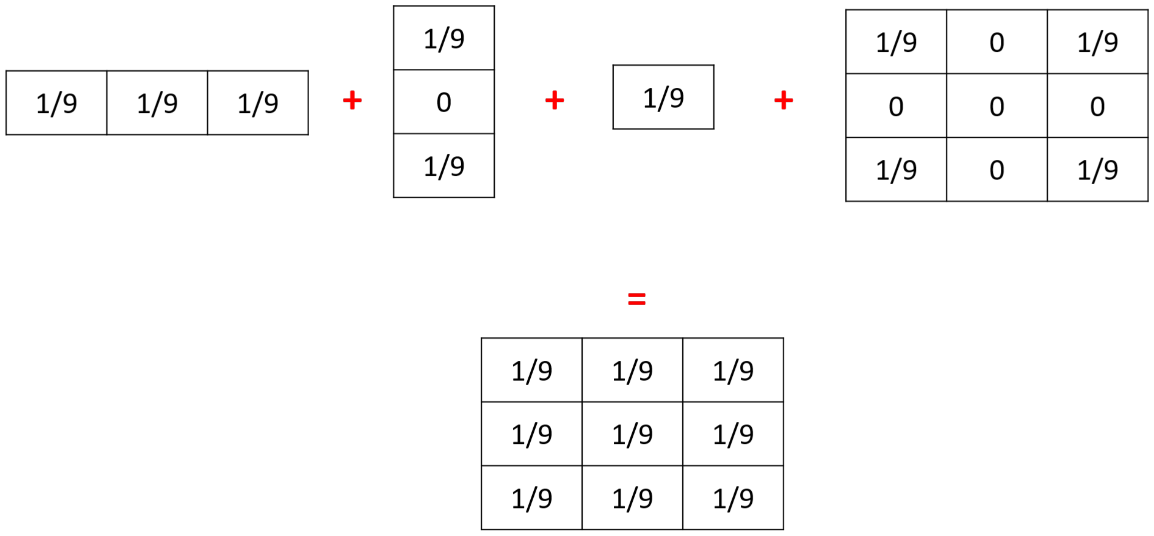


Figure 48.8: resources/images/recipes/addition_matrix.png

Diffusion with parameters

Sometimes writing diffusion matrix is not exactly what you want, and you may prefer to just give some parameters to compute the correct diffusion matrix. You can use the following facets in order to do that : `propagation`, `variation` and `radius`.

Depending on which `propagation` you choose, and how many neighbors your grid have, the propagation matrix will be compute differently. The propagation matrix will have the size : range*2+1.

Let's note **P** for the propagation value, **V** for the variation, **R** for the range and **N** for the number of neighbors.

- **With diffusion propagation**

For diffusion propagation, we compute following the following steps:

(1) We determine the "minimale" matrix according to N (if N = 8, the matrix will be `[[P/9,P/9,P/9][P/9,1/9,P/9][P/9,P/9,P/9]]`. if N = 4, the matrix will be `[[0,P/5,0][P/5,1/5,P/5][0,P/5,0]]`).

(2) If R != 1, we propagate the matrix R times to obtain a `[2*R+1][2*R+1]` matrix (same computation as for `cycle_length`).

(3) If V != 0, we substract each value by V*DistanceFromCenter (DistanceFromCenter depends on N).

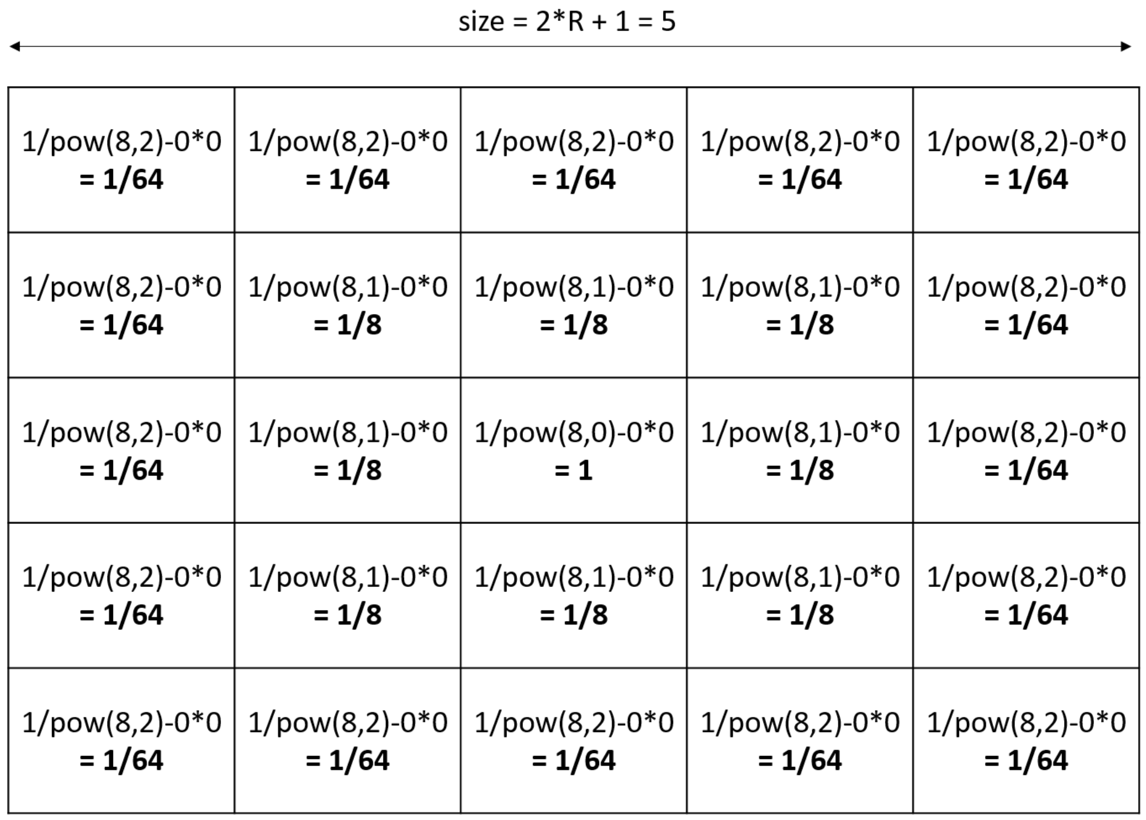
Ex with the default values (P=1, R=1, V=0, N=8):

- **With gradient propagation**

The value of each cell will be equal to **P/POW(N,DistanceFromCenter)-DistanceFromCenter*V**. (DistanceFromCenter depends on N).

Ex with R=2, other parameters default values (R=2, P=1, V=0, N=8):

Note that if you declared a diffusion matrix, you cannot use those 3 facets (it will raise a warning). Note also that if you use parameters, you will only have uniform matrix.



size = 2*R + 1 = 5

1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**
1/pow(8,2)-0*0 **= 1/64**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,2)-0*0 **= 1/64**
1/pow(8,2)-0*0 **= 1/64**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,0)-0*0 **= 1**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,2)-0*0 **= 1/64**
1/pow(8,2)-0*0 **= 1/64**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,1)-0*0 **= 1/8**	1/pow(8,2)-0*0 **= 1/64**
1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**	1/pow(8,2)-0*0 **= 1/64**

Figure 48.9: resources/images/recipes/gradient_computation_from_parameters.png

Computation methods

You can compute the output matrix using two computation methods by using the facet `method` : the dot product and the convolution. Note that the result of those two methods is exactly the same (except if you use the `avoid_mask` facet, the results can be slightly differents between the two computations).

Convolution

`convolution` is the default computation method for the diffusion. For every output cells, we will multiply the input values and the flipped kernel together, as shown in the following image :

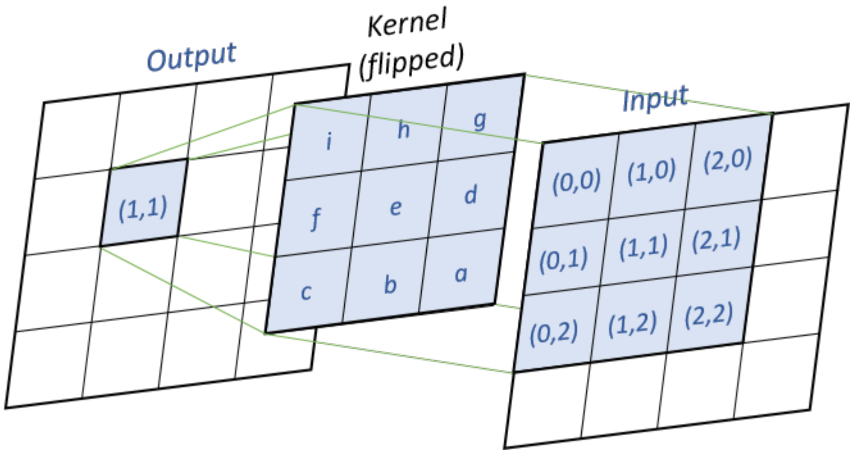


Figure 48.10: resources/images/recipes/convolution.png

Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
  for (j = 0 ; j < y.nbCols ; j++)
    for (m = 0 ; m < k.nbRows ; m++)
      for (n = 0 ; n < k.nbCols ; n++)
        y[i,j] += k[k.nbRows - m - 1, k.nbCols - n - 1]
            * x[i - k.nbRows/2 + m, j - k.nbCols/2 + n]
```

Dot Product

`dot_product` method will compute the matrix using a simple dot product between the matrix. For every input cells, we multiply the cell by the kernel matrix, as shown in the following image :

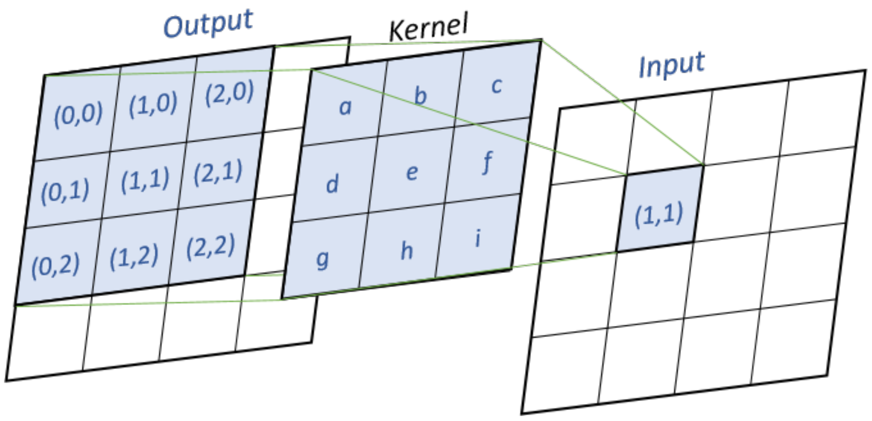


Figure 48.11: resources/images/recipes/dot_product.png

Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
  for (j = 0 ; j < y.nbCols ; j++)
    for (m = 0 ; m < k.nbRows ; m++)
      for (n = 0 ; n < k.nbCols ; n++)
        y[i - k.nbRows/2 + m, j - k.nbCols/2 + n] += k[m, n] * x[
    i, j]
```

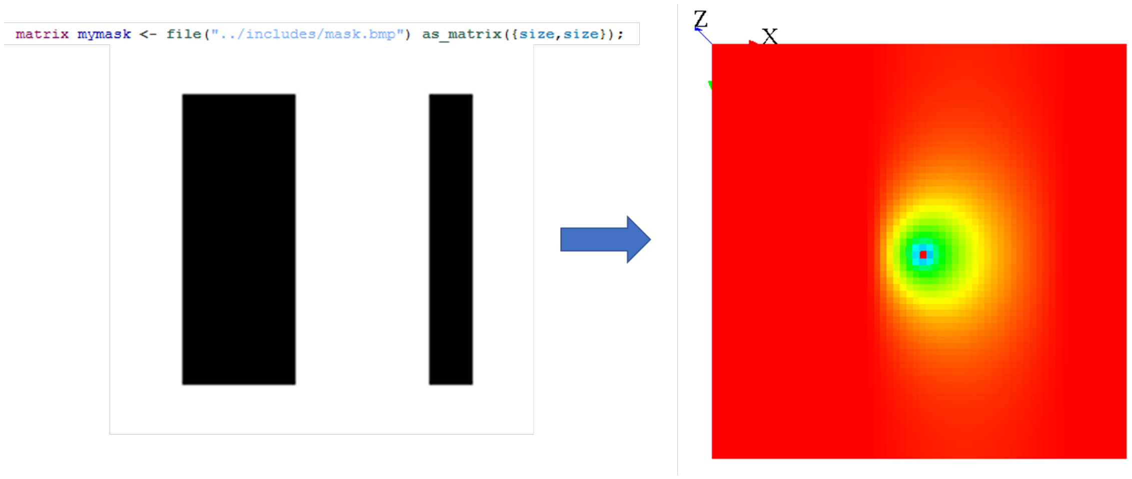
Using mask

Generalities

If you want to propagate some values in an heterogeneous grid, you can use some mask to forbid some cells to propagate their values.

You can pass a matrix to the facet `mask`. All the values smaller than `-1` will not propagate, and all the values greater or equal to `-1` will propagate.

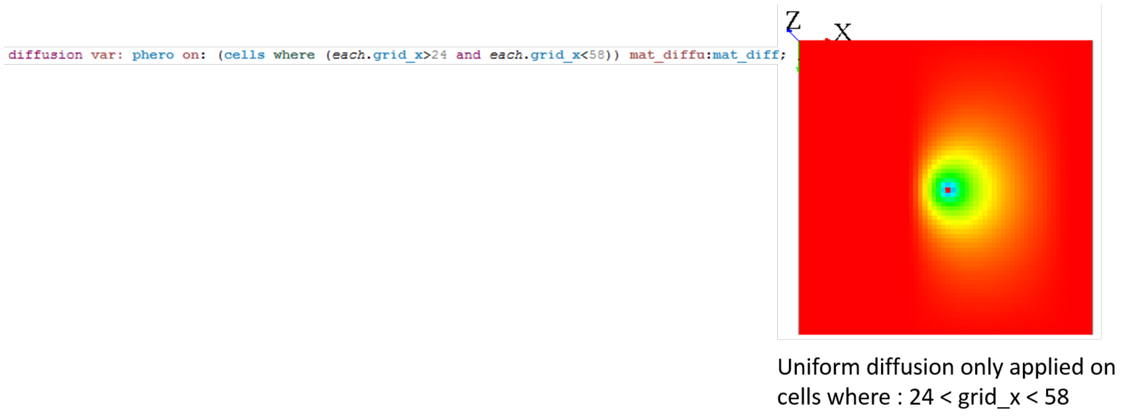
A simple way to use mask is by loading an image :



```
matrix mymask <- file("../includes/mask.bmp") as_matrix({size,size});
```

Figure 48.12: resources/images/recipes/simple_mask.png

Note that when you use the `on` facet for the `diffuse` statement, you can choose only some cells, and not every cells. In fact, when you restrain the values to be diffuse, it is exactly the same process as if you were defining a mask.



```
diffusion var: phero on: (cells where (each.grid_x>24 and each.grid_x<58)) mat_diffu:mat_diff;
```

Uniform diffusion only applied on cells where : 24 < grid_x < 58

Figure 48.13: resources/images/recipes/mask_with_on_facet.png

When your diffusion is combined with a mask, the default behavior is that the non-masked cells will diffuse their values in **all** existing cells (that means, even the masked cells !). To change this behavior, you can use the facet `avoid_mask`. In that case, the value which was supposed to be affected to the masked cell will be redistributed to the neighboring non-masked cells.

Tips

Masks can be used to simulate a lot of environments. Here are some ideas for your models:

Wall blocking the diffusion

If you want to simulate a wall blocking a uniform diffusion, you can declare a second diffusion matrix that will be applied only on the cells where your wall will be. This diffusion matrix will "push" the values outside from himself, but conserving the values (the sum of the values of the diffusion still have to be equal to 1) :

```
matrix<float> mat_diff <- matrix([
            [1/9,1/9,1/9],
            [1/9,1/9,1/9],
            [1/9,1/9,1/9]]);

matrix<float> mat_diff_left_wall <- matrix([
            [0.0,0.0,2/9],
            [0.0,0.0,4/9],
            [0.0,0.0,2/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_x>30)) matrix:
   mat_diff;
    diffuse var: phero on: (cells where(each.grid_x=30)) matrix:
   mat_diff_left_wall;
}
```

Note that almost the same result can be obtained by using the facet `avoid_mask` : the value of all masked cells will remain at 0, and the value which was supposed to be affected to the masked cell will be distributed to the neighboring cells. Notice that the results can be slightly different if you are using the `convolution` or the `dot_product` method : the algorithm of redistribution of the value to the neighboring cells is a bit different. We advise you to use the `dot_product` with the `avoid_mask` facet, the results are more accurates.

Wind pushing the diffusion

Let's simulate a uniform diffusion that is pushed by a wind from "north" everywhere in the grid. A wind from "west" as blowing at the top side of the grid. We will here have to build 2

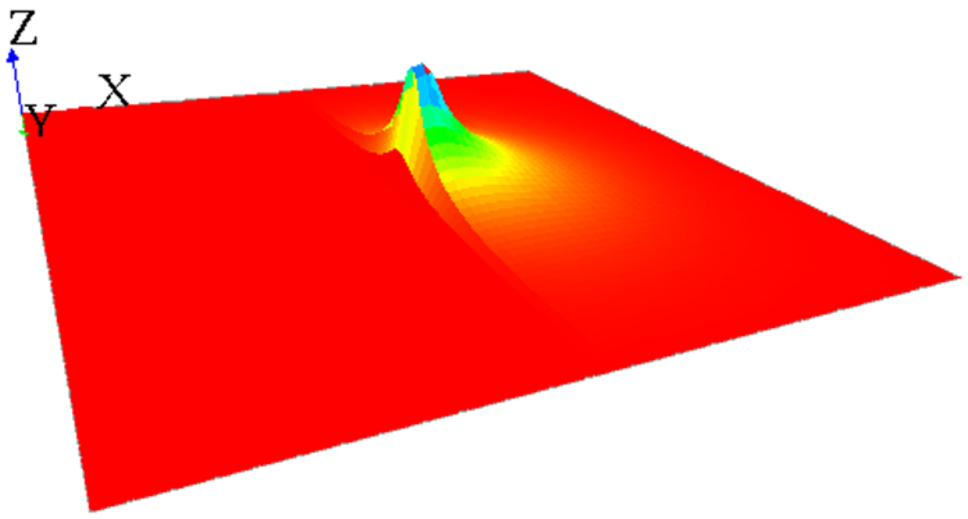


Figure 48.14: resources/images/recipes/wall_simulation.png

matrix : one for the uniform diffusion, one for the "north" wind and one for the "west" wind. The sum of the values for the 2 matrix meant to simulate the wind will be equal to 0 (as it will be add to the diffusion matrix).

```
matrix<float> mat_diff <- matrix([
        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);

matrix<float> mat_wind_from_west <- matrix([
        [-1/9,0.0,1/9],
        [-1/9,0.0,1/9],
        [-1/9,0.0,1/9]]);

matrix<float> mat_wind_from_north <- matrix([
        [-1/9,-1/9,-1/9],
        [0.0,0.0,0.0],
        [1/9,1/9,1/9]]);

reflex diff {
    diffuse var: phero on: cells matrix:mat_diff;
    diffuse var: phero on: cells matrix:mat_wind_from_north;
    diffuse var: phero on: (cells where (each.grid_y>=32)) matrix
   :mat_wind_from_west;
}
```

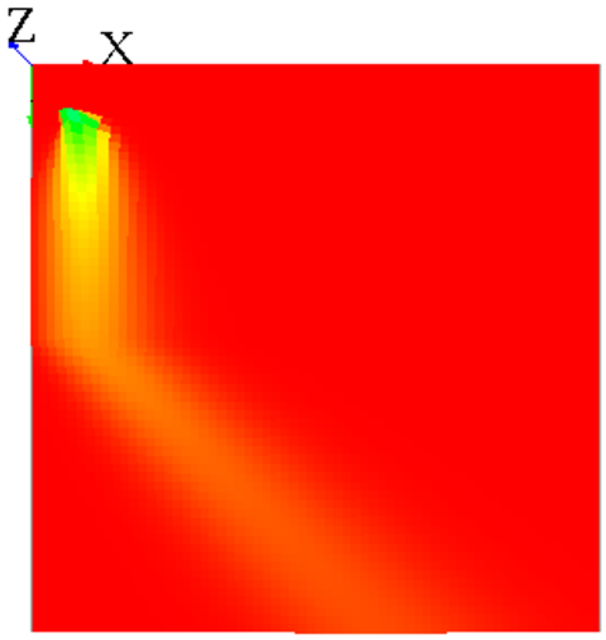


Figure 48.15: resources/images/recipes/diffusion_with_wind.png

Endless world

Note that when your world is not a torus, it has the same effect as a *mask*, since all the values outside from the world cannot diffuse some values back :

You can "fake" the fact that your world is endless by adding a different diffusion for the cells with `grid_x=0` to have almost the same result :

```
matrix<float> mat_diff <- matrix([
             [1/9,1/9,1/9],
             [1/9,1/9,1/9],
             [1/9,1/9,1/9]]);
```

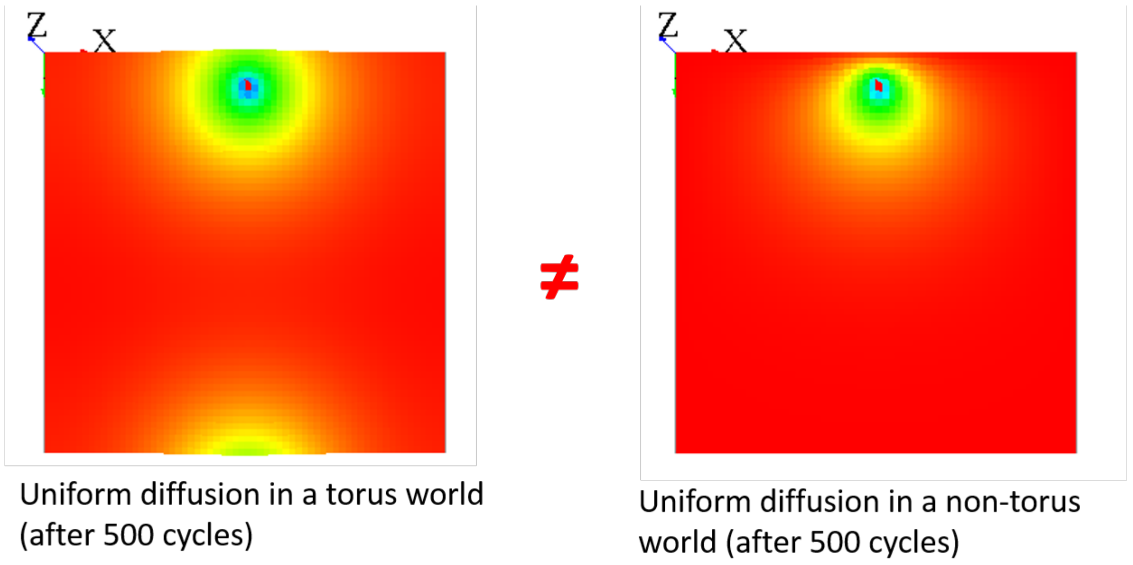

Z
X
≠
Uniform diffusion in a torus world (after 500 cycles)
Uniform diffusion in a non-torus world (after 500 cycles)

Figure 48.16: resources/images/recipes/uniform_diffusion_near_edge.png

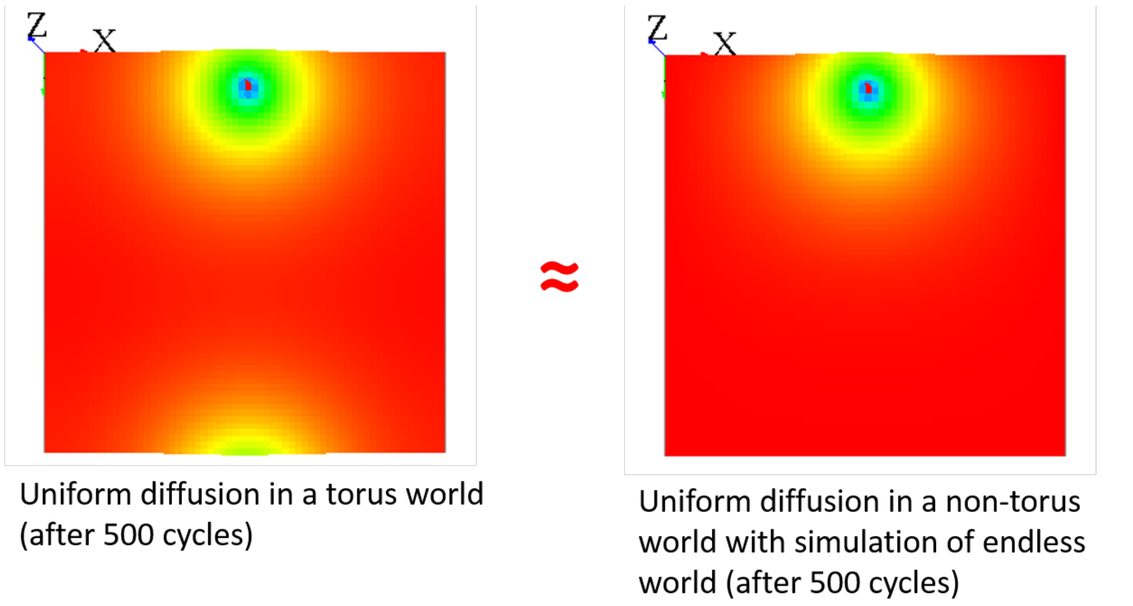

Z
X
≈
Uniform diffusion in a torus world (after 500 cycles)
Uniform diffusion in a non-torus world with simulation of endless world (after 500 cycles)

Figure 48.17: resources/images/recipes/uniform_diffusion_near_edge_with_mask.png

```
matrix<float> mat_diff_upper_edge <- matrix([
            [0.0,0.0,0.0],
            [1/9+7/81,2/9+1/81,1/9+7/81],
            [1/9,1/9,1/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_y>0)) matrix:
   mat_diff;
    diffuse var: phero on: (cells where(each.grid_y=0)) matrix:
   mat_diff_upper_edge;
}
```

Pseudo code

This section is more for a better understanding of the source code.

Here is the pseudo code for the computation of diffusion :

1) : Execute the statement `diffuse`, store the diffusions in a map (from class *Diffusion-Statement* to class *GridDiffuser*) :

```
- Get all the facet values
- Compute the "real" mask, from the facet "mask:" and the facet "
   on:".
  - If no value for "mask:" and "on:" all the grid, the mask is
   equal to null.
- Compute the matrix of diffusion
  - If no value for "matrix:", compute with "nb_neighbors", "
   is_gradient", "proportion", "propagation", "variation", "range
   ".
  - Then, compute the matrix of diffusion with "cycle_length".
- Store the diffusion properties in a map
  - Map : ["method_diffu", "is_gradient", "matrix", "mask", "
   min_value"] is value, ["var_diffu", "grid_name"] is key.
  - If the key exists in the map, try to "mix" the diffusions
    - If "method_diffu", "mask" and "is_gradient" equal for the 2
    diffusions, mix the diffusion matrix.
```

2) : At the end of the step, execute the diffusions (class *GridDiffuser*) :

```
- For each key of the map,
  - Load the couple "var_diffu" / "grid_name"
  - Build the "output" and "input" array with the dimension of
   the grid.
  - Initialize the "output" array with -Double.MAX_VALUE.
  - For each value of the map for that key,
    - Load all the properties : "method_diffu", "is_gradient", "
   matrix", "mask", "min_value"
    - Compute :
      - If the cell is not masked, if the value of input is >
   min_value, diffuse to the neighbors.
        - If the value of the cell is equal to -Double.MAX_VALUE,
    remplace it by input[idx] * matDiffu[i][j].
        - Else, do the computation (gradient or diffusion).
    - Finish the diffusion :
      - If output[idx] > -Double.MAX_VALUE, write the new value
   in the cell.
```

Chapter 49

Using Database Access

Database features of GAMA provide a set of actions on Database Management Systems (DBMS) and Multi-Dimensional Database for agents in GAMA. Database features are implemented in the irit.gaml.exxtensions.database plug-in with these features: * Agents can execute SQL queries (create, Insert, select, update, drop, delete) to various kinds of DBMS. * Agents can execute MDX (Multidimensional Expressions) queries to select multidimensional objects, such as cubes, and return multidimensional cellsets that contain the cube's data . These features are implemented in two kinds of component: *skills* (SQLSKILL, MDXSKILL) and agent (AgentDB)

SQLSKILL and AgentDB provide almost the same features (a same set of actions on DBMS) but with certain slight differences:

- An agent of species AgentDB will maintain a unique connection to the database during the whole simulation. The connection is thus initialized when the agent is created.
- In contrast, an agent of a species with the SQLSKILL skill will open a connection each time he wants to execute a query. This means that each action will be composed of three running steps:
  - Make a database connection.
  - Execute SQL statement.
  - Close database connection. > An agent with the SQLSKILL spends lot of time to create/close the connection each time it needs to send a query; it saves the database connection (DBMS often limit the number of simultaneous connections). In contrast, an AgentDB agent only needs to establish one database connection and it can be used for any actions. Because it does not need to create and

close database connection for each action: therefore, actions of AgentDB agents are executed faster than actions of SQLSKILL ones but we must pay a connection for each agent.

- With an inheritance agent of species AgentDB or an agent of a species using SQLSKILL, we can query data from relational database for creating species, defining environment or analyzing or storing simulation results into RDBMS. On the other hand, an agent of species with MDXKILL supports the OLAP technology to query data from data marts (multidimensional database). The database features help us to have more flexibility in management of simulation models and analysis of simulation results.

Description

- **Plug-in**: *irit.gaml.extensions.database*
- **Author**: TRUONG Minh Thai, Frederic AMBLARD, Benoit GAUDOU, Christophe SIBERTIN-BLANC

Supported DBMS

The following DBMS are currently supported: * SQLite * MySQL Server * PostgreSQL Server * SQL Server * Mondrian OLAP Server * SQL Server Analysis Services

Note that, other DBMSs require a dedicated server to work while SQLite on only needs a file to be accessed. All the actions can be used independently from the chosen DBMS. Only the connection parameters are DBMS-dependent.

SQLSKILL

Define a species that uses the SQLSKILL skill

Example of declaration:

```
entities {
   species toto skills: [SQLSKILL]
   {
     //insert your descriptions here
```

```
    }
    ...
}
```

Agents with such a skill can use additional actions (defined in the skill)

Map of connection parameters for SQL

In the actions defined in the SQLSkill, a parameter containing the connection parameters is required. It is a map with the following *key::value* pairs:

Key	Optional	Description
dbtype	No	DBMS type value. Its value is a string. We must use “mysql” when we want to connect to a MySQL. That is the same for “postgres”, “sqlite” or “sqlserver” (ignore case sensitive)
host	Yes	Host name or IP address of data server. It is absent when we work with SQlite.
port	Yes	Port of connection. It is not required when we work with SQLite.
database	No	Name of database. It is the file name including the path when we work with SQLite.
user	Yes	Username. It is not required when we work with SQLite.
passwd	Yes	Password. It is not required when we work with SQLite.
srid	Yes	srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in *Preferences->External* configuration.

Table 1: Connection parameter description

Example: Definitions of connection parameter

```
// POSTGRES connection parameter
map <string, string>  POSTGRES <- [
      'host'::'localhost',
      'dbtype'::'postgres',
      'database'::'BPH',
     'port'::'5433',
     'user'::'postgres',
     'passwd'::'abc'];

//SQLite
map <string, string>  SQLITE <- [
    'dbtype'::'sqlite',
    'database'::'../includes/meteo.db'];

// SQLSERVER connection parameter
map <string, string> SQLSERVER <- [
    'host'::'localhost',
    'dbtype'::'sqlserver',
    'database'::'BPH',
    'port'::'1433',
    'user'::'sa',
    'passwd'::'abc'];

// MySQL connection parameter
map <string, string>  MySQL <- [
    'host'::'localhost',
    'dbtype'::'MySQL',
    'database'::'', // it may be a null string
    'port'::'3306',
    'user'::'root',
    'passwd'::'abc'];
```

Test a connection to database

Syntax: > ***testConnection*** *(params: connection_parameter)* The action tests the connection to a given database. * **Return**: boolean. It is: * *true*: the agent can connect to the DBMS (to the given Database with given name and password) * *false*: the agent cannot connect * **Arguments**: * *params*: (type = map) map of connection parameters * **Exceptions**: *GamaRuntimeException*

Example: Check a connection to MySQL

```
if (self testConnection(params:MySQL)){
    write "Connection is OK" ;
}else{
    write "Connection is false" ;
}
```

Select data from database

Syntax: > ***select*** *(param: connection_parameter, select: selection_string,values: value_list)* The action creates a connection to a DBMS and executes the select statement. If the connection or selection fails then it throws a GamaRuntimeException. * **Return**: list < list >. If the selection succeeds, it returns a list with three elements: * The first element is a list of column name. * The second element is a list of column type. * The third element is a data set. * **Arguments**: * *params*: (type = map) map containing the connection parameters * *select*: (type = string) select string. The selection string can contain question marks. * *values*: List of values that are used to replace question marks in appropriate. This is an optional parameter. * **Exceptions**: *GamaRuntimeException*

Example: select data from table points

```
map <string, string>   PARAMS <- ['dbtype'::'sqlite', 'database
   '::'../includes/meteo.db'];
list<list> t <- list<list> (self select(params:PARAMS,
                                   select:"SELECT * FROM points ;"));
```

Example: select data from table point with question marks from table points

```
map <string, string>   PARAMS <- ['dbtype'::'sqlite', 'database
   '::'../includes/meteo.db'];
list<list> t <- list<list> (self select(params: PARAMS,
                                                select: "SELECT
   temp_min FROM points where (day>? and day<?);"
                                                values: [10,20] ));
```

Insert data into database

Syntax: > **_insert** (param: connection_parameter, into: table_name, columns: column_list, values: value'_list)*The action creates a connection to a DBMS and executes the insert statement. If the connection or insertion fails then it throws a_GamaRuntimeException*. * **Return**: int > If the insertion succeeds, it returns a number of records inserted by the insert. * **Arguments**: params*: (type = map) map containing the connection parameters.*

into: (type = string) table name. columns*: (type=list) list of column names of table. It is an optional argument. If it is not applicable then all columns of table are selected. values*: (type=list) list of values that are used to insert into table corresponding to columns. Hence the columns and values must have same size. * **Exceptions**:_GamaRuntimeException

Example: Insert data into table registration

```
map<string, string> PARAMS <- ['dbtype'::'sqlite', 'database
   '::'../../includes/Student.db'];

do insert (params: PARAMS,
              into: "registration",
              values: [102, 'Mahnaz', 'Fatma', 25]);

do insert (params: PARAMS,
               into: "registration",
               columns: ["id", "first", "last"],
               values: [103, 'Zaid tim', 'Kha']);

int n <- insert (params: PARAMS,
                       into: "registration",
                      columns: ["id", "first", "last"],
                      values: [104, 'Bill', 'Clark']);
```

Execution update commands

Syntax: > ***executeUpdate*** *(param: connection_parameter, updateComm: table_name, values: value_list)* The action executeUpdate executes an update command (create/insert/delete/drop) by using the current database connection of the agent. If the database connection does not exist or the update command fails then it throws a GamaRuntimeException. Otherwise it returns an integer value. * **Return**: int. If the insertion succeeds, it returns a number of records inserted by the insert. * **Arguments**: * *params*: (type = map) map containing the connection parameters * *updateComm*: (type = string) SQL command string. It may be commands: *create*, *update*, *delete* and *drop* with or without question marks. * *columns*: (type=list) list of column names of table. * *values*: (type=list) list of values that are used to replace question marks if appropriate. This is an optional parameter. * **Exceptions**: *GamaRuntimeException*

Examples: Using action executeUpdate do sql commands (create, insert, update, delete and drop).

```
map<string, string> PARAMS <- ['dbtype'::'sqlite',  'database
   '::'../../includes/Student.db'];
// Create table
do executeUpdate (params: PARAMS,
                           updateComm: "CREATE TABLE
   registration"
                                              + "(id INTEGER
   PRIMARY KEY, "
                                              + " first TEXT NOT
   NULL, " + " last TEXT NOT NULL, "
                                              + " age INTEGER);");

// Insert into
do executeUpdate (params: PARAMS ,
                          updateComm: "INSERT INTO
   registration " + "VALUES(100, 'Zara', 'Ali', 18);");
do insert (params: PARAMS, into: "registration",
               columns: ["id", "first", "last"],
               values: [103, 'Zaid tim', 'Kha']);

// executeUpdate with question marks
do executeUpdate (params: PARAMS,
                          updateComm: "INSERT INTO
   registration " + "VALUES(?, ?, ?, ?);" ,
                          values: [101, 'Mr', 'Mme', 45]);

//update
int n <-  executeUpdate (params: PARAMS,
                                   updateComm: "UPDATE
   registration SET age = 30 WHERE id IN (100, 101)" );

// delete
int n <- executeUpdate (params: PARAMS,
                                  updateComm: "DELETE FROM
   registration where id=? ",
                                  values: [101] );

// Drop table
do executeUpdate (params: PARAMS, updateComm: "DROP TABLE
   registration");
```

MDXSKILL

MDXSKILL plays the role of an OLAP tool using select to query data from OLAP server to GAMA environment and then species can use the queried data for any analysis purposes. ### Define a species that uses the MDXSKILL skill Example of declaration:

```
entities {
    species olap skills: [MDXSKILL]
     {
        //insert your descriptions here

     }
      ...
```

Agents with such a skill can use additional actions (defined in the skill)

Map of connection parameters for MDX

In the actions defined in the SQLSkill, a parameter containing the connection parameters is required. It is a map with following key::value pairs:

Key	Optional	Description
olaptype	No	OLAP Server type value. Its value is a string. We must use "SSAS/XMLA" when we want to connect to an SQL Server Analysis Services by using XML for Analysis. That is the same for "MONDRIAN/XML" or "MONDRIAN" (ignore case sensitive)
dbtype	No	DBMS type value. Its value is a string. We must use "mysql" when we want to connect to a MySQL. That is the same for "postgres" or "sqlserver" (ignore case sensitive)
host	No	Host name or IP address of data server.
port	No	Port of connection. It is no required when we work with SQLite.
database	No	Name of database. It is file name include path when we work with SQLite.
catalog	Yes	Name of catalog. It is an optional parameter. We do not need to use it when we connect to SSAS via XMLA and its file name includes the path when we connect a ROLAP database directly by using Mondrian API (see Example as below)

Key	Optional	Description
user	No	Username.
passwd	No	Password.

Table 2: OLAP Connection parameter description

Example: Definitions of OLAP connection parameter

```
//Connect SQL Server Analysis Services via XMLA
    map<string,string> SSAS <- [
                'olaptype'::'SSAS/XMLA',
                'dbtype'::'sqlserver',
                'host'::'172.17.88.166',
                'port'::'80',
                'database'::'olap',
                'user'::'test',
                'passwd'::'abc'];

//Connect Mondriam server via XMLA
    map<string,string>  MONDRIANXMLA <- [
                'olaptype'::"MONDRIAN/XMLA",
                'dbtype'::'postgres',
                'host'::'localhost',
                'port'::'8080',
                'database'::'MondrianFoodMart',
                'catalog'::'FoodMart',
                'user'::'test',
                'passwd'::'abc'];

//Connect a ROLAP server using Mondriam API
    map<string,string>  MONDRIAN <- [
                'olaptype'::'MONDRIAN',
                'dbtype'::'postgres',
                'host'::'localhost',
                'port'::'5433',
                'database'::'foodmart',
                'catalog'::'../includes/FoodMart.xml',
                'user'::'test',
                                'passwd'::'abc'];
```

Test a connection to OLAP database

Syntax: > ***testConnection*** *(params: connection_parameter)* The action tests the connection to a given OLAP database. * **Return**: boolean. It is: * *true*: the agent can connect to the DBMS (to the given Database with given name and password) * *false*: the agent cannot connect * **Arguments**: * *params*: (type = map) map of connection parameters * **Exceptions**: *GamaRuntimeException*

Example: Check a connection to MySQL

```
if (self testConnection(params:MONDIRANXMLA)){
    write "Connection is OK";
}else{
    write "Connection is false";
}
```

Select data from OLAP database

Syntax: > ***select*** *(param: connection_parameter, onColumns: column_string, onRows: row_string from: cube_string, where: condition_string, values: value_list)* The action creates a connection to an OLAP database and executes the select statement. If the connection or selection fails then it throws a *GamaRuntimeException*. * **Return**: list < list >. If the selection succeeds, it returns a list with three elements: * The first element is a list of column name. * The second element is a list of column type. * The third element is a data set. * **Arguments**: * *params*: (type = map) map containing the connection parameters * *onColumns*: (type = string) declare the select string on columns. The selection string can contain question marks. * *onRows*: (type = string) declare the selection string on rows. The selection string can contain question marks. * *from*: (type = string) specify cube where data is selected. The cube_string can contain question marks. * where_: (type = string) specify the selection conditions. The condiction_string can contains question marks. This is an optional parameter. values*: List of values that are used to replace question marks if appropriate. This is an optional parameter.* **Exceptions**:_GamaRuntimeException

Example: select data from SQL Server Analysis Service via XMLA

```
if (self testConnection[ params::SSAS]){
    list l1  <- list(self select (params: SSAS ,
        onColumns: " { [Measures].[Quantity], [Measures].[Price]
   }",
        onRows:" { { { [Time].[Year].[All].CHILDREN } * "
        + " { [Product].[Product Category].[All].CHILDREN } * "
        +"{ [Customer].[Company Name].&[Alfreds Futterkiste], "
        +"[Customer].[Company Name].&[Ana Trujillo Emparedadosy
   helados], "
        + "[Customer].[Company Name].&[Antonio Moreno Taquería] }
    } } " ,
        from : "FROM [Northwind Star] "));
    write "result1:"+ l1;
}else {
    write "Connect error";
}
```

Example: select data from Mondrian via XMLA with question marks in selection

```
if (self testConnection(params:MONDRIANXMLA)){
    list<list> l2  <- list<list> (self select(params:
   MONDRIANXMLA,
    onColumns:" {[Measures].[Unit Sales], [Measures].[Store Cost
   ], [Measures].[Store Sales]} ",
    onRows:"  Hierarchize(Union(Union(Union({([Promotion Media].[
   All Media],"
    +" [Product].[All Products])}, "
    +" Crossjoin([Promotion Media].[All Media].Children, "
    +" {[Product].[All Products]})), "
    +" Crossjoin({[Promotion Media].[Daily Paper, Radio, TV]}, "
    +" [Product].[All Products].Children)), "
    +" Crossjoin({[Promotion Media].[Street Handout]}, "
    +" [Product].[All Products].Children))  ",
    from:" from [?]  " ,
    where :" where [Time].[?] " ,
    values:["Sales",1997]));
    write "result2:"+ l2;
}else {
    write "Connect error";
}
```

AgentDB

AgentBD is a built-in species, which supports behaviors that look like actions in SQLSKILL but differs slightly with SQLSKILL in that it uses only one connection for several actions. It means that AgentDB makes a connection to DBMS and keeps that connection for its later operations with DBMS. ### Define a species that is an inheritance of agentDB Example of declaration:

```
entities {
    species agentDB parent: AgentDB {
     {
        //insert your descriptions here
     }
       ...
}
```

Connect to database

Syntax:

> ***Connect*** *(param: connection_parameter)* This action makes a connection to DBMS. If a connection is established then it will assign the connection object into a built-in attribute of species (conn) otherwise it throws a GamaRuntime-Exception. * **Return**: connection * **Arguments**: * *params*: (type = map) map containing the connection parameters * **Exceptions**: GamaRuntimeException **Example**: Connect to PostgreSQL

```
// POSTGRES connection parameter
map <string, string>  POSTGRES <- [
                                        'host'::'localhost',
                                        'dbtype'::'postgres',
                                        'database'::'BPH',
                                        'port'::'5433',
                                        'user'::'postgres',
                                        'passwd'::'abc'];
ask agentDB {
      do connect (params: POSTGRES);
}
```

Check agent connected a database or not

Syntax:

> ***isConnected*** *(param: connection_parameter)* This action checks if an agent is connecting to database or not. * **Return**: Boolean. If agent is connecting to a database then isConnected returns true; otherwise it returns false. * **Arguments**: * *params*: (type = map) map containing the connection parameters

Example: Using action executeUpdate do sql commands (create, insert, update, delete and drop).

```
ask agentDB {
    if (self isConnected){
              write "It already has a connection";
    }else{
              do connect (params: POSTGRES);
        }
}
```

Close the current connection

Syntax:

> ***close*** This action closes the current database connection of species. If species does not has a database connection then it throws a GamaRuntimeException. * **Return**: null If the current connection of species is close then the action return null value; otherwise it throws a GamaRuntimeException.

Example:

```
ask agentDB {
    if (self isConnected){
          do close;
    }
}
```

Get connection parameter

Syntax:

> ***getParameter*** This action returns the connection parameter of species. * **Return**: map < string, string >

Example:

```
ask agentDB {
    if (self isConnected){
        write "the connection parameter: " +(self getParameter);
        }
}
```

Set connection parameter

Syntax:

> ***setParameter*** *(param: connection_parameter)* This action sets the new values for connection parameter and closes the current connection of species. If it can not close the current connection then it will throw GamaRuntimeException. If the species wants to make the connection to database with the new values then action connect must be called. * **Return**: null * **Arguments**: * *params*: (type = map) map containing the connection parameters * **Exceptions**: *GamaRuntimeException*

Example:

```
ask agentDB {
    if (self isConnected){
              do setParameter(params: MySQL);
              do connect(params: (self getParameter));
        }
}
```

Retrieve data from database by using AgentDB

Because of the connection to database of AgentDB is kept alive then AgentDB can execute several SQL queries with only one connection. Hence AgentDB can do actions such as **select**, **insert**, **executeUpdate** with the same parameters of those actions of SQLSKILL *except* ***params*** *parameter is always absent*.

Examples:

```
map<string, string> PARAMS <- ['dbtype'::'sqlite', 'database
   '::'../../includes/Student.db'];
ask agentDB {
   do connect (params: PARAMS);
   // Create table
   do executeUpdate (updateComm: "CREATE TABLE registration"
    + "(id INTEGER PRIMARY KEY, "
        + " first TEXT NOT NULL, " + " last TEXT NOT NULL, "
        + " age INTEGER);");
   // Insert into
   do executeUpdate ( updateComm: "INSERT INTO registration "
        + "VALUES(100, 'Zara', 'Ali', 18);");
   do insert (into: "registration",
     columns: ["id", "first", "last"],
     values: [103, 'Zaid tim', 'Kha']);
   // executeUpdate with question marks
   do executeUpdate (updateComm: "INSERT INTO registration VALUES
   (?, ?, ?, ?);",
     values: [101, 'Mr', 'Mme', 45]);
   //select
    list<list> t <- list<list> (self select(
     select:"SELECT * FROM registration;"));
    //update
    int n <-  executeUpdate (updateComm: "UPDATE registration SET
    age = 30 WHERE id IN (100, 101)");
     // delete
     int n <- executeUpdate ( updateComm: "DELETE FROM
   registration where id=? ",  values: [101] );
     // Drop table
      do executeUpdate (updateComm: "DROP TABLE registration");
}
```

Using database features to define environment or create species

In Gama, we can use results of select action of SQLSKILL or AgentDB to create species or define boundary of environment in the same way we do with shape files. Further more, we can also save simulation data that are generated by simulation including geometry data to database.

Define the boundary of the environment from database

- **Step 1**: specify select query by declaration a map object with keys as below:

Key	Optional	Description
dbtype	No	DBMS type value. Its value is a string. We must use "mysql" when we want to connect to a MySQL. That is the same for "postgres", "sqlite" or "sqlserver" (ignore case sensitive)
host	Yes	Host name or IP address of data server. It is absent when we work with SQlite.
port	Yes	Port of connection. It is not required when we work with SQLite.
database	No	Name of database. It is the file name including the path when we work with SQLite.
user	Yes	Username. It is not required when we work with SQLite.
passwd	Yes	Password. It is not required when we work with SQLite.
srid	Yes	srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in Preferences->External configuration.
select	No	Selection string

Table 3: Select boundary parameter description

Example:

```
map<string,string> BOUNDS <- [
    //'srid'::'32648',
    'host'::'localhost',
        'dbtype'::'postgres',
    'database'::'spatial_DB',
    'port'::'5433',
        'user'::'postgres',
    'passwd'::'tmt',
    'select'::'SELECT ST_AsBinary(geom) as geom FROM bounds;' ];
```

- **Step 2**: define boundary of environment by using the map object in first step.

```
geometry shape <- envelope(BOUNDS);
```

Note: We can do the same way if we work with MySQL, SQLite, or SQLServer and we must convert Geometry format in GIS database to binary format.

Create agents from the result of a select action

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1**: Define a species with SQLSKILL or AgentDB

```
entities {
    species toto skills: SQLSKILL {
     {
        //insert your descriptions here
     }
   ...
}
```

- **Step 2**: Define a connection and selection parameters

```
global {
    map<string,string>  PARAMS <- ['dbtype'::'sqlite','database
   '::'../includes/bph.sqlite'];
    string location <- 'select ID_4, Name_4, ST_AsBinary(geometry
   ) as geom from vnm_adm4
                                    where id_2=38253 or id_2
   =38254;';
    ...
}
```

- **Step 3**: Create species by using selected results

```
init {
   create toto {
      create locations from: list(self select (params: PARAMS,
                                                        select
   : LOCATIONS))
                                 with:[ id:: "id_4",
   custom_name:: "name_4", shape::"geom"];
    }
   ...
}
```

Save Geometry data to database

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1**: Define a species with SQLSKILL or AgentDB

```
entities {
    species toto skills: SQLSKILL {
     {
        //insert your descriptions here

     }
        ...
}
```

- **Step 2**: Define a connection and create GIS database and tables

```
global {
    map<string,string> PARAMS <-  ['host'::'localhost', 'dbtype
   '::'Postgres', 'database'::'',
                                                                   'port
   '::'5433', 'user'::'postgres', 'passwd'::'tmt'];

    init {
        create toto ;
        ask toto {
            if (self testConnection[ params::PARAMS]){
                // create GIS database
                do executeUpdate(params:PARAMS,
                        updateComm: "CREATE DATABASE spatial_db
   with TEMPLATE = template_postgis;");
                    remove key: "database" from: PARAMS;
                put "spatial_db" key:"database" in: PARAMS;
                //create table
                           do executeUpdate params: PARAMS
                  updateComm : "CREATE TABLE buildings "+
                  "( "   +
                                  " name character varying(255),
    " +
                                      " type character varying
   (255), " +
                                      " geom GEOMETRY " +
                                  ")";
            }else {
                write "Connection to MySQL can not be established
    ";
            }
        }
    }
}
```

- **Step 3**: Insert geometry data to GIS database

```
ask building {
   ask DB_Accessor {
    do insert(params: PARAMS,
                        into: "buildings",
            columns: ["name", "type","geom"],
            values: [myself.name,myself.type,myself.shape];
   }
}
```

Chapter 50

Calling R

Introduction

R language is one of powerful data mining tools, and its community is very large in the world (See the website: http://www.r-project.org/). Adding the R language into GAMA is our strong endeavors to accelerate many statistical, data mining tools into GAMA.

RCaller 2.0 package (Website: http://code.google.com/p/rcaller/) is used for GAMA 1.6.1.

Table of contents

- Load the package:
- Read data from iris:
- Build the decision tree:
- Build the random forest of 50 decision trees:
- Predict the acceptance of test set:
- Calculate the accuracy: * Output * Calling R codes from a text file (.R, .txt) WITH the parameters * Example 4 * Mean.R file * Output * Example 5 * AddParam.R file * Output

Configuration in GAMA

1) Install R language into your computer.

2) In GAMA, select menu option: **Edit/Preferences**.

3) In "**Config RScript's path**", browse to your "**Rscript**" file (R language installed in your system).

Notes: Ensure that install.packages("Runiversal") is already applied in R environment.

Calling R from GAML

Calling the built-in operators

Example 1

```
model CallingR

global {
    list X <- [2, 3, 1];
    list Y <- [2, 12, 4];

    list result;

    init{
        write corR(X, Y); // -> 0.755928946018454
        write meanR(X); // -> 2.0
    }
}
```

Calling R codes from a text file (.R,.txt) WITHOUT the parameters

Using **R_compute(String RFile)** operator. This operator DOESN'T ALLOW to add any parameters form the GAML code. All inputs is directly added into the R codes. **Remarks**: Don't let any white lines at the end of R codes. **R_compute** will return the last variable of R file, this parameter can be a basic type or a list. Please ensure that the called packages must be installed before using.

Example 2

```
model CallingR

global
{
    list result;

    init{
        result <- R_compute("C:/YourPath/Correlation.R");
        write result at 0;
    }
}
```

Above syntax is deprecated, use following syntax with R_file instead of R_compute:

```
model CallingR

global
{
    file result;

    init{
        result <- R_file("C:/YourPath/Correlation.R");
        write result.contents;
    }
}
```

Correlation.R file

```
x <- c(1, 2, 3)

y <- c(1, 2, 4)

result <- cor(x, y, method = "pearson")
```

Output

```
result::[0.981980506061966]
```

Example 3

```
model CallingR

global
{
    list result;

    init{
        result <- R_compute("C:/YourPath/RandomForest.R");

        write result at 0;
    }
}
```

RandomForest.R file

```
# Load the package:

library(randomForest)
```

```
# Read data from iris:

data(iris)

nrow<-length(iris[,1])

ncol<-length(iris[1,])

idx<-sample(nrow,replace=FALSE)

trainrow<-round(2*nrow/3)

trainset<-iris[idx[1:trainrow],]

# Build the decision tree:

trainset<-iris[idx[1:trainrow],]

testset<-iris[idx[(trainrow+1):nrow],]

# Build the random forest of 50 decision trees:

model<-randomForest(x= trainset[,-ncol], y= trainset[,ncol], mtry
   =3, ntree=50)

# Predict the acceptance of test set:

pred<-predict(model, testset[,-ncol], type="class")

# Calculate the accuracy:

acc<-sum(pred==testset[, ncol])/(nrow-trainrow)
```

Output

```
acc::[0.98]
```

Calling R codes from a text file (.R, .txt) WITH the parameters

Using **R_compute_param(String RFile, List vectorParam)** operator. This operator ALLOWS to add the parameters from the GAML code.

Remarks: Don't let any white lines at the end of R codes. **R_compute_param** will return the last variable of R file, this parameter can be a basic type or a list. Please ensure that the called packages must be installed before using.

Example 4

```
model CallingR

global
{
    list X <- [2, 3, 1];
    list result;

    init{
        result <- R_compute_param("C:/YourPath/Mean.R", X);
        write result at 0;
    }
}
```

Mean.R file

```
result <- mean(vectorParam)
```

Output

```
result::[3.3333333333333]
```

Example 5

```
model CallingR

global {
    list X <- [2, 3, 1];
    list result;

        init{
        result <- R_compute_param("C:/YourPath/AddParam.R", X);
        write result at 0;
    }
}
```

AddParam.R file

```
v1 <- vectorParam[1]

v2<-vectorParam[2]

v3<-vectorParam[3]

result<-v1+v2+v3
```

Output

`result::[10]` //: # (endConcept|call_r)

Chapter 51

Using FIPA ACL

The communicating skill offers some actions and built-in variables which enable agents to communicate with each other using the FIPA interaction protocol. This document describes the built-in variables and actions of this skill. Examples are found in the models library bundled with GAMA.

Variables

- **accept_proposals (list)**: A list of 'accept_proposal' performative messages of the agent's mailbox having .
- **agrees (list)**: A list of 'accept_proposal' performative messages.
- **cancels (list)**: A list of 'cancel' performative messages.
- **cfps (list)**: A list of 'cfp' (call for proposal) performative messages.
- **conversations (list)**: A list containing the current conversations of agent. Ended conversations are automatically removed from this list.
- **failures (list)**: A list of 'failure' performative messages.
- **informs (list)**: A list of 'inform' performative messages.
- **messages (list)**: The mailbox of the agent, a list of messages of all types of performatives.
- **proposes (list)**: A list of 'propose' performative messages .
- **queries (list)**: A list of 'query' performative messages.
- **refuses (list)**: A list of 'propose' performative messages.
- **reject_proposals (list)**: A list of 'reject_proposals' performative messages.
- **requests (list)**: A list of 'request' performative messages.

- **requestWhens (list)**: A list of 'request-when' performative messages.
- **subscribes (list)**: A list of 'subscribe' performative messages.

Actions

accept_proposal

Replies a message with an 'accept_proposal' performative message * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

agree

Replies a message with an 'agree' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

cancel

Replies a message with a 'cancel' peformative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

cfp

Replies a message with a 'cfp' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

end_conversation

Replies a message with an 'end_conversation' peprformative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsible of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

failure

Replies a message with a 'failure' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

inform

Replies a message with an 'inform' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

propose

Replies a message with a 'propose' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

query

Replies a message with a 'query' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

refuse

Replies a message with a 'refuse' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

reject_proposal

Replies a message with a 'reject_proposal' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

reply

Replies a message. This action should be only used to reply a message in a 'no-protocol' conversation and with a 'user defined performative'. For performatives supported by GAMA (i.e., standard FIPA performatives), please use the 'action' with the same name of 'performative'. For example, to reply a message with a 'request' performative message, the modeller should use the 'request' action. * returns: unknown * message (message): The message to be replied * performative (string): The performative of the replying message * content (list): The content of the replying message

request

Replies a message with a 'request' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

send

Starts a conversation/interaction protocol. * returns: msi.gaml.extensions.fipa.Message * receivers (list): A list of receiver agents * content (list): The content of the message. A list of any GAML type * performative (string): A string, representing the message performative * protocol (string): A string representing the name of interaction protocol

start_conversation

Starts a conversation/interaction protocol. * returns: msi.gaml.extensions.fipa.Message * receivers (list): A list of receiver agents * content (list): The content of the message. A list of any GAML type * performative (string): A string, representing the message performative * protocol (string): A string representing the name of interaction protocol

subscribe

Replies a message with a 'subscribe' performative message. * returns: unknown * message (message): The message to be replied * content (list): The content of the replying message

Chapter 52

Using GAMAnalyzer

Install

Go to Git View -> Click on Import Projects Add the dependencies in ummisco.gama.feature.dependencies

GamAnalyzer is a tool to monitor several multi-agents simulation

The “agent_group_follower” goal is to monitor and analyze a group of agent during several simulation. This group of agent can be chosen by the user according to criteria chosen by the user. The monitoring process and analysis of these agents involves the extraction, processing and visualization of their data at every step of the simulation. The data for each simulation are pooled and treated commonly for their graphic representation or clusters.

Built-in Variable

- **varmap**: All variable that can be analyzed or displayed in a graph.
- **numvarmap**: Numerical variable (on this variable all the aggregator numeric are computed).
- **qualivarmap**: All non numerical variable. Could be used for BDI to analyze beliefs.
- **metadatahistory**: See updateMetaDataHistory. This matrice store all the metadata like getSimulationScope(), getClock().getCycle(), getUniqueSimName(scope), rule,

scope.getAgentScope().getName(), this.getName(), this.agentsCourants.copy(scope), this.agentsCourants.size(), this.getGeometry().

- **lastdetailedvarvalues**: store all the value (in varmap) for all the followed agent for the last iteration.
- **averagehistory**: Average value for each of the numvar
- **stdevhistory**: Std deviation value for each of the numvar
- **minhistory**: Min deviation value for each of the numvar
- **maxhistory**: Max deviation value for each of the numvar
- **distribhistoryparams**: Gives the interval of the distribution described in distribhistory
- **distribhistory**: Distribution of numvarmap
- **multi_metadatahistory**: Aggregate each metadatahistory for each experiment

Example

This example is based on a toy model which is only composed of wandering people. In this example we will use GamAnalyzer to follow the agent people.

```
agent_group_follower peoplefollower;
```

```
create agentfollower
{
  do analyse_cluster species_to_analyse:"people";
  peoplefollower<-self;
}
```

expGlobalNone

No clustering only the current agent follower is displayed

```
aspect base {
  display_mode <-"global";
  clustering_mode <-"none";
  draw shape color: #red;
}
```

expSimGlobalNone

The agent_group_follower corresponding to the current iteration and all the already launch experiments are displayed.

```
aspect simglobal{
  display_mode <-"simglobal";
  clustering_mode <-"none";
  draw shape color: #red;
  int curColor <-0;
  loop geom over: allSimShape{
    draw geom color:SequentialColors[curColor] at:{location.x,
   location.y,curColor*10};
    curColor <- curColor+1;
  }
}
```

expCluster

The agent group follower is divided in cluster computed thanks to a dbscan algorithm. Only the current agent_group_follower is displayed

```
aspect cluster {
  display_mode <-"global";
  clustering_mode <-"dbscan";
  draw shape color: #red;
}
```

expClusterSimGlobal

The agent_group_follower (made of different cluster) corresponding to the current iteration and all the already launch experiments are displayed.

```
aspect clusterSimGlobal {
  display_mode <-"simglobal";
  clustering_mode <-"dbscan";
  draw shape color: #red;
  int curColor <-0;
  loop geom over: allSimShape{
    draw geom color:SequentialColors[curColor] at:{location.x,
   location.y,curColor*10};
    curColor <- curColor+1;
  }
}
```

Chapter 53

Using BDI

Install

You need to run the Git version.

The plugin need to be add with Eclipse doing the following:

- In ummisco.gama.feature.core open the feature.xml file.
- In plug-ins click add the msi.gaml.architecture.simplebdi

Acteur Projet

A website (still in construction) of the ACTEUR project can be found here http://acteur-anr.fr/

An introduction to cognitive agent

The belief-desire-intention software model (usually referred to simply, but ambiguously, as BDI) is a software model developed for programming intelligent agents.

- **Belief**: State of the agent.
- **Desire**: Objectives that the agent would like to accomplish.

- **Intention**: What the agent has chosen to do.
  - **Plan**: Sequences of actions that an agent can perform to achieve one or more of its intensions.

Basic Example: A fire rescue model using cognitive agent

We introduce a simple example to illustrate the use of the BDI architecture.

This simple model consists in creating "cognitive" agent whose goal is to extinguish a fire. In a first approximation we consider only one static water area and fire area. The aim is not to have a realistic model but to illustrate how to give a "cognitive" behavior to an agent using the BDI architecture.

First let's create a BDI agent using the key control **simple_bdi** (A description of all existing control architectures is available here.)

Species Helicopter creation

```
species helicopter skills:[moving] control: simple_bdi{
...
}
```

Attributes

The species `helicopter` needs 2 attributes to represent the water value and its speed.

```
float waterValue;
float speed <- 10.0;
```

Predicates

The predicate are the structure that are used to define a belief, a desire or an intention. In this model we choose to declare 3 different predicates.

```
predicate patrol_desire <- new_predicate("patrol") with_priority
   1;
predicate water_predicate <- new_predicate("has water", true)
   with_priority 3;
predicate no_water_predicate <- new_predicate("has water", false)
    ;
```

The **new_predicate()** tool creates a predicate. It needs a name (string type) and it can contain a map of values, a priority (double type) or a truth value (boolean type). The **with_-priority** tool add a priority to a predicate. The priority is used as an argument when the agent has to choose between two predicates (to choose an intention for example).

Initialization

The initialization consists in setting the attribute **waterValue** to 1 and to add one desire. Three optional parameters are also set. The first desire added in the desire base is the **patrol_desire** saying that the helicopter wants to patrol. The optional parameters are specific to the BDI plug-in. You can specify the commitment of an agent to his intentions and his plans with the variables intention_persistence and plan_persistence that are floats between 0.0 (no commitment) and 1.0. The variable probabilistic_choice is a boolean that enables the agent to use a probabilistic choice (when true) or a deterministic choice (when false) when trying to find a plan or an intention.

```
waterValue <-1.0;
do add_desire(patrol_desire);
intention_persistence <- 1.0;
plan_persistence <- 1.0;
probabilistic_choice <- false;
```

Perception

At each iteration, the helicopter has two perceptions to do. The first one is about itself. The helicopter needs to perceive if it has water or not. If it has water, it adds the belief corresponding belief and removes the belief that it does not have water. And if it does not have water, that is the contrary.

```
perceive target:self{
    if(waterValue>0){
        do add_belief(water_predicate);
        do remove_belief(no_water_predicate);
    }
    if(waterValue<=0){
        do add_belief(no_water_predicate);
        do remove_belief(water_predicate);
    }
}
```

The second perception is about the fires. Here, the fires are represented with the species **fireArea**. The helicopter has a radius of perception of 10 meters. If it perceives a fire, it will focus on the location of this fire. The **focus** tool create a belief with the same name as the focus (here, "fireLocation") and will store the value of the focused variable (here, the variable location from the specie fireArea) with a priority of 10 in this example. Once the fire is perceived, the helicopter removes its intention of patrolling.

```
perceive target:fireArea in: 10{
    focus fireLocation var:location priority:10;
    ask myself{
        do remove_intention(patrol_desire, true);
    }
}
```

Rules

The agent can use rules to create desires from beliefs. In this example, the agent has two rules. The first **rule** is to have a desire corresponding to the belief of a location of a fire. It means that when the agent has the belief that there is a fire in a particular location, it will have the desire to extinguish it. This permits to have the location value in the desire base. The second rule is to create the desire to have water when the agent has the belief that it not has water.

```
rule belief: new_predicate("fireLocation") new_desire:
   get_belief_with_name("fireLocation");
rule belief: no_water_predicate new_desire: water_predicate;
```

Plan

Patrolling

This plan will be used when the agent has the intention to patrol.

```
plan patrolling intention: patrol_desire{
  do wander;
}
```

stopFire

This plan is executed when the agent has the intention to extinguish a fire.

```
plan stopFire intention: new_predicate("fireLocation") {
     point target_fire <- point(get_current_intention().values["
    location_value"] );
     if(waterValue>0){
         if (self distance_to target_fire <= 1) {
             fireArea current_fire <- fireArea first_with (each.
    location = target_fire);
             if (current_fire != nil) {
                  waterValue <- waterValue - 1.0;
                  current_fire.size <-  current_fire.size - 1;
                  if ( current_fire.size <= 0) {
                     ask  current_fire {do die;}
                     do remove_belief(get_current_intention());
                     do remove_intention(get_current_intention(),
    true);
                     do add_desire(patrol_desire);
                 }
             } else {
                 do remove_belief(get_current_intention());
                 do remove_intention(get_current_intention(), true
    );
                 do add_desire(patrol_desire);
             }
         } else {
             do goto target: target_fire;
         }
     } else {
         do add_subintention(get_current_intention(),
    water_predicate,true);
         do current_intention_on_hold();
     }
}
```

gotoTakeWater

This plan is executed when the agent has the intention to have water.

```
plan gotoTakeWater intention: water_predicate {
        waterArea wa <- first(waterArea);
        do goto target: wa);
        if (self distance_to wa <= 1) {
            waterValue <- waterValue + 2.0;
    }
}
```

Plans can have other options. They can have a priority (with the facet priority), a boolean condition to start (with the facet when) or a boolean condition to stop (with the facet finished_when).

Rest of the code

Aspect of the helicopter

```
aspect base {
    draw circle(1) color: #black;
}
```

FireArea Species

```
species fireArea{
        float size <-1.0;

        aspect base {
          draw circle(size) color: #red;
        }
}
```

WaterArea Species

```
species waterArea{
    float size <-10.0;

    aspect base {
      draw circle(size) color: #blue;
    }
}
```

Chapter 54

Advanced Driving Skill

This page aims at presenting how to use the advanced driving skill in models.

The use of the advanced driving skill requires to use 3 skills: * **Advanced driving skill**: dedicated to the definition of the driver species. It provides the driver agents with variables and actions allowing to move an agent on a graph network and to tune its behavior. * **Road skill**: dedicated to the definition of roads. It provides the road agents with variables and actions allowing to registers agents on the road. * **RoadNode skill**: dedicated to the definition of node. It provides the node agents with variables allowing to take into account the intersection of roads and the traffic signals.

Table of contents

Structure of the network: road and roadNode skills

The advanced driving skill is versatile enough to be usable with most of classic road GIS data, in particular OSM data. We use a classic format for the roads and nodes. Each road is a polyline composed of road sections (segments). Each road has a target node and a source node. Each node knows all its input and output roads. A road is considered as directed. For bidirectional roads, 2 roads have to be defined corresponding to both directions. Each road will be the **linked_road** of the other. Note that for some GIS data, only one road is defined for bidirectional roads, and the nodes are not explicitly defined. In this case, it is very easy, using the GAML language, to create the reverse roads and the corresponding nodes (it only requires few lines of GAML).

images/roads_structure.PNG

A lane can be composed of several lanes and the vehicles will be able to change at any time its lane. Another property of the road that will be taken into account is the maximal authorized speed on it. Note that even if the user of the plug-in has no information about these values for some of the roads (the OSM data are often incomplete), it is very easy using the GAML language to fill the missing value by a default value. It is also possible to change these values dynamically during the simulation (for example, to take into account that after an accident, a lane of a road is closed or that the speed of a road is decreased by the authorities).

images/roads.PNG

The **road skill** provides the road agents with several variables that will define the road properties: * **lanes**: integer, number of lanes. * **maxspeed**: float; maximal authorized speed on the road. * **linked_road**: road agent; reverse road (if there is one). * **source_node**: node agent; source node of the road. * **target_node**: node agent; target node of the road.

It provides as well the road agents with one read only variable: * **agents_on**: list of list (of driver agents); for each lane, the list of driver agents on the road.

The*roadNode skill**provides the road agents with several variables that will define the road properties:** *roads_in**: list of road agents; the list of road agents that have this node for target node.*** roads_out**: list of road agents; the list of road agents that have this node for source node.** *stop**: list of list of road agents; list of stop signals, and for each stop signal, the list of concerned roads.**

It provides as well the road agents with one read only variable:

- **block**: map: key: driver agent, value: list of road agents; the list of driver agents blocking the node, and for each agent, the list of concerned roads.

Advanced driving skill

Each driver agent has a planned trajectory that consists in a succession of edges. When the driver agent enters a new edge, it first chooses its lane according to the traffic density, with a bias for the rightmost lane. The movement on an edge is inspired by the Intelligent Driver Model. The drivers have the possibility to change their lane at any time (and not only when entering a new edge).

The **advanced driving skill** provides the driver agents with several variables that will define the car properties and the personality of the driver: * **final_target**: point; final location that the agent wants to reach (its goal). * **vehicle_length**: float; length of the vehicle. * **max_acceleration**: float; maximal acceleration of the vehicle. * **max_speed**: float; maximal speed of the vehicle. * **right_side_driving**: boolean; do drivers drive on the right side of the road? * **speed_coef**: float; coefficient that defines if the driver will try to drive above or below the speed limits. * **security_distance_coeff**: float; coefficient for the security distance. The security distance will depend on the driver speed and on this coefficient. * **proba_lane_change_up**: float; probability to change lane to a upper lane if necessary (and if possible). * **proba_lane_change_down**: float; probability to change lane to a lower lane if necessary (and if possible). * **proba_use_linked_road**: float; probability to take the reverse road if necessary (if there is a reverse road). * **proba_respect_priorities**: float; probability to respect left/right (according to the driving side) priority at intersections. * **proba_respect_stops**: list of float; probabilities to respect each type of stop signals (traffic light, stop sign...). * **proba_block_node**: float; probability to accept to block the intersecting roads to enter a new road.

It provides as well the driver agents with several read only variables:

- **speed**: float; speed expected according to the road **max_value**, the car properties, the personality of the driver and its **real_speed**.
- **real_speed**: float; real speed of the car (that takes into account the other drivers and the traffic signals).
- **current_path**: path (list of roads to follow); the path that the agent is currently following.
- **current_target**: point; the next target to reach (sub-goal). It corresponds to a node.
- **targets**: list of points; list of locations (sub-goals) to reach the final target.
- **current_index**: integer; the index of the current goal the agent has to reach.
- **on_linked_road**: boolean; is the agent on the linked road?

Of course, the values of these variables can be modified at any time during the simulation. For example, the probability to take a reverse road (**proba_use_linked_road**) can be increased if the driver is stucked for several minutes behind a slow vehicle.

In addition, the advanced driving skill provides the driver agents with several actions: * **compute_path**: arguments: a graph and a target node. This action computes from a graph the shortest path to reach a given node. * **drive**: no argument. This action moves the driver on its current path according to the traffic condition and the driver properties (vehicle properties and driver personality).

the **drive** action works as follow: while the agent has the time to move (**remaining_time** > **0**), it first defines the speed expected. This speed is computed from the **max_speed** of the road, the current **real_speed**, the **max_speed**, the **max_acceleration** and the **speed_coef** of the driver (see Equation 1).

```
speed_driver = Min(max_speed_driver , Min(real_speed_driver +
   max_acceleration_driver ,max_speed_road * speed_coef_driver))
```

Then, the agent moves toward the current target and compute the remaining time. During the movement, the agents can change lanes (see below). If the agent reaches its final target, it stops; if it reaches its current target (that is not the final target), it tests if it can cross the intersection to reach the next road of the current path. If it is possible, it defines its new target (target node of the next road) and continues to move.

images/drive_action.png

The function that defines if the agent crosses or not the intersection to continue to move works as follow: first, it tests if the road is blocked by a driver at the intersection (if the road is blocked, the agent does not cross the intersection). Then, if there is at least one stop signal at the intersection (traffic signal, stop sign…), for each of these signals, the agent tests its probability to respect or not the signal (note that the agent has a specific probability to respect each type of signals). If there is no stopping signal or if the agent does not respect it, the agent checks if there is at least one vehicle coming from a right (or left if the agent drives on the left side) road at a distance lower than its security distance. If there is one, it tests its probability to respect this priority. If there is no vehicle from the right roads or if it chooses to do not respect the right priority, it tests if it is possible to cross the intersection to its target road without blocking the intersection (i.e. if there is enough space in the target road). If it can cross the intersection, it crosses it; otherwise, it tests its probability to block the node: if the agent decides nevertheless to cross the intersection, then the perpendicular roads will be blocked at the intersection level (these roads will be unblocked when the agent is going to move).

images/stop_at_intersection.png

Concerning the movement of the driver agents on the current road, the agent moves from a section of the road (i.e. segment composing the polyline) to another section according to the maximal distance that the agent can moves (that will depend on the remaining time).

For each road section, the agent first computes the maximal distance it can travel according the remaining time and its speed. Then, the agent computes its security distance according to its speed and its **security_distance_coeff**. While its remaining distance is not null, the agent computes the maximal distance it can travel (and the corresponding lane), then it moves according to this distance (and update its current lane if necessary). If the agent is not blocked by another vehicle and can reach the end of the road section, it updates its current road section and continues to move.

images/follow_driving.png

The computation of the maximal distance an agent can move on a road section consists in computing for each possible lane the maximal distance the agent can move. First, if there is a lower lane, the agent tests the probability to change its lane to a lower one. If it decides to test the lower lane, the agent computes the distance to the next vehicle on this lane and memorizes it. If this distance corresponds to the maximal distance it can travel, it chooses this lane; otherwise it computes the distance to the next vehicle on its current lane and memorizes it if it is higher than the current memorized maximal distance. Then if the memorized distance is lower than the maximal distance the agent can travel and if there is an upper lane, the agents tests the probability to change its lane to a upper one. If it decides to test the upper lane, the agent computes the distance to the next vehicle on this lane and memorizes it if it is higher than the current memorized maximal distance. At last, if the memorized distance is still lower than the maximal distance it can travel, if the agent is on the highest lane and if there is a reverse road, the agent tests the probability to use the reverse road (linked road). If it decides to use the reverse road, the agent computes the distance to the next vehicle on the lane 0 of this road and memorizes the distance if it is higher than the current memorized maximal distance.

images/define_max_dist.png

Application example

We propose a simple model to illustrate the driving skill. We define a driver species. When a driver agent reaches its destination, it just chooses a new random final target. In the same way, we did not define any specific behavior to avoid traffic jam for the driver agents: once they compute their path (all the driver agents use for that the same road graph with the same weights), they never re-compute it even if they are stucked in a traffic jam. Concerning the traffic signals, we just consider the traffic lights (without any pre-processing: we consider the raw OSM data). One step of the simulation represents 1 second. At last, in order to clarify the explanation of the model, we chose to do not present the parts of the GAML code that concern the simulation visualization.

images//sim_snapshot.png

The following code shows the definition of species to represent the road infrastructure:

```
species road skills: [skill_road] {
  string oneway;
}

species node skills: [skill_road_node] {
  bool is_traffic_signal;
  int time_to_change <- 100;
  int counter <- rnd (time_to_change) ;

  reflex dynamic when: is_traffic_signal {
    counter <- counter + 1;
    if (counter >= time_to_change) {
      counter <- 0;
      stop[0] <-empty(stop[0])? roads_in : [];
    }
  }
}
```

In order to use our driving skill, we just have to add the **skill_road_node** to the **node** species and the **skill_road** to the **road** species. In addition, we added to the road species a variable called **oneway** that will be initialized from the OSM data and that represents the traffic direction (see the OSM map features for more details). Concerning the node, we defined 3 new attributes:

- **is_traffic_signal**: boolean; is the node a traffic light?
- **time_to_change**: integer; represents for the traffic lights the time to pass from the red light to the green light (and vice versa).
- **counter**: integer; number of simulation steps since the last change of light color (used by the traffic light nodes).

In addition, we defined for the **node** species a reflex (behavior) called **dynamic** that will be activated only for traffic light nodes and that will increment the **counter** value. If this counter is higher than **time_to_change**, this variable is set to 0, and the node change the value of the **stop** variable: if the traffic light was green (i.e. there is no road concerns by this stop sign), the list of block roads is set by all the roads that enter the node; if the traffic light was red (i.e. there is at least one road concerns by this stop sign), the list of block roads is set to an empty list.

The following code shows the definition of driver species:

```
species driver skills: [advanced_driving] {
  reflex time_to_go when: final_target = nil {
    current_path <- compute_path(
        graph: road_network, target: one_of(node));
  }
  reflex move when: final_target != nil {
    do drive;
  }
}
```

In order to use our driving plug-in, we just have to add the **advanced_driving** to the **driver** species. For this species, we defined two reflexes: * **time_to_go**: activated when the agent has no final target. In this reflex, the agent will randomly choose one of the nodes as its final target, and computed the path to reach this target using the * **road_network** graph. Note that it will have been possible to take into account the knowledge that each agent has concerning the road network by defining a new variable of type map (dictionary) containing for each road a given weight that will reflect the driver knowledge concerning the network (for example, the known traffic jams, its favorite roads....) and to use this map for the path computation. * **move**: activated when the agent has a final target. In this reflex, the agent will drive in direction of its final target.

We describe in the following code how we initialize the simulation:

```
init {
  create node from: file("nodes.shp") with:[
    is_traffic_signal::read("type")="traffic_signals"];

  create road from: file("roads.shp")
    with:[lanes::int(read("lanes")),
    maxspeed::float(read("maxspeed")),
    oneway::string(read("oneway"))]
    {
      switch oneway {
        match "no" {
          create road {
            lanes <- myself.lanes;
            shape <- polyline(reverse
              (myself.shape.points));
            maxspeed <- myself.maxspeed;
            linked_road <- myself;
            myself.linked_road <- self;
          }
        }
        match "-1" {
          shape <- polyline(reverse(shape.points));
        }
      }
    }
  }
  map general_speed_map <-  road as_map
    (each::(each.shape.perimeter/(each.maxspeed)));

  road_network <-  (as_driving_graph(road, node))
     with_weights general_speed_map;

  create driver number: 10000 {
    location <- one_of(node).location;
    vehicle_length <- 3.0;
    max_acceleration <- 0.5 + rnd(500) / 1000;
    speed_coeff <- 1.2 - (rnd(400) / 1000);
    right_side_driving <- true;
    proba_lane_change_up <- rnd(500) / 500;
    proba_lane_change_down <- 0.5+ (rnd(250) / 500);
    security_distance_coeff <- 3 - rnd(2000) / 1000);
    proba_respect_priorities <- 1.0 - rnd(200/1000);
    proba_respect_stops <- [1.0 - rnd(2) / 1000];
    proba_block_node <- rnd(3) / 1000;
    proba_use_linked_road <- rnd(10) / 1000;
  }
}
```

In this code, we create the node agents from the node shapefile (while reading the attributes contained in the shapefile), then we create in the same way the road agents. However, for the road agents, we use the **oneway** variable to define if we should or not reverse their geometry (**oneway** = "-1") or create a reverse road (**oneway** = "no"). Then, from the road and node agents, we create a graph (while taking into account the **maxspeed** of the road for the weights of the edges). This graph is the one that will be used by all agents to compute their path to their final target. Finally, we create 1000 driver agents. At initialization, they are randomly placed on the nodes; their vehicle has a length of 3m; the maximal acceleration of their vehicle is randomly drawn between 0.5 and 1; the speed coefficient of the driver is randomly drawn between 0.8 and 1.2; they are driving on the right side of the road; their probability of changing lane for a upper lane is randomly drawn between 0 and 1.0; their probability of changing lane for a lower lane is randomly drawn between 0.5 and 1.0; the security distance coefficient is randomly drawn between 1 and 3; their probability to respect priorities is randomly drawn between 0.8 and 1; their probability to respect light signal is randomly drawn between 0.998 and 1; their probability to block a node is randomly drawn between 0 and 0.003; their probability to use the reverse road is randomly drawn between 0 and 0.01;

The complete code of the model with the data can be found here

Chapter 55

Manipulate Dates

Managing Time in Models

If some models are based on a abstract time - only the number of cycles is important - others are based on a real time. In order to manage the time, GAMA provides some tools to manage time.

First, GAMA allows to define the duration of a simulation step. It provides access to different time variables. At last, since GAMA 1.7, it provides a date variable type and some global variables allowing to use a real calendar to manage time.

Definition of the step and use of temporal unity values

GAMA provides three important global variables to manage time:

- `cycle` (int - not modifiable): the current simulation step - this variable is incremented by 1 at each simulation step
- `step` (float - can be modified): the duration of a simulation step (in seconds). By default the duration is one second.
- `time` (float - not modifiable): the current time spent since the beginning of the simulation - this variable is computed at each simulation step by: time = cycle * step.

The value of the cycle and time variables are shown in the top left (green rectangle) of the simulation interface. Clicking on the green rectangle allows to display either the number cycles or the time variable. Concerning this variable, it is presented following a years - month - days - hours - minutes - seconds format. In this presentation, every months are considered as being composed of 30 days (the different number of days of months are not taken into account).

Concerning the step facet, the variable can be modified by the modeler. A classic way of doing it consists in reediting the variable in the global section:

```
global {
	float step <- 1 #hour;
}
```

In this example, each simulation step will represent 1 hour. This time will be taken into account for all actions based on time (e.g. moving actions).

Note that the value of the step variable should be given in seconds. To facilitate the definition of the step value and of all expressions based on time, GAMA provides different built-in constant variables accessible with the "#" symbol:

- `#s` : second - 1 second
- `#mn` : minute - 60 seconds
- `#hour` : hour - 60 minutes - 3600 seconds
- `#day` : day - 24 hours - 86400 seconds
- `#month` : month - 30 days - 2592000 seconds
- `#year` : year - 12 month - 3.1104E7

The date variable type and the use of a real calendar

Since GAMA 1.7, it is possible to use a real calendar to manage the time. For that, the modeler have just to define the starting date of the simulation. This variable is of type date which allow to represent a date and time. A date variable has several attributes:

- `year` (int): the year component of the date
- `month` (int): the month component of the date
- `day` (int): the day component of the date
- `hour` (int): the hour component of the date
- `minute` (int): the minute component of the date
- `second` (int): the second component of the date

- `day_of_week` (int): the day of the week
- `week_of_year` (int): the week of the year

Several ways can be used to define a date. The simplest consists in using a list of int values: [year,month of the year,day of the month, hour of the day, minute of the hour, second of the minute]

```
date my_date <- date([2010,3,23,17,30,10]); // the 23th of March
   2010, at 17:30:10
```

Another way consists in using a string with the good format:

```
date my_date <- date("2010-3-23T17:30:10+07:00");
```

Note that the current date can be access through the #now built-in variable (variable of type date).

In addition, GAMA provides different useful operators working on dates. For instance, it is possible to compute the duration in seconds between 2 dates using the "-" operator. The result is given in seconds:

```
float d <- starting_date - my_date;
```

It is also possible to add or subtract a duration (in seconds) to a date:

```
write "my_date + 10: " + (my_date + 10);
write "my_date - 10: " + (my_date - 10);
```

At last, it is possible to add or subtract a duration (in years, months, weeks, days, hours, minutes, seconds) to a date:

```
write "my_date add_years 1: " + (my_date add_years 1);
write "my_date add_months 1: " + (my_date add_months 1);
write "my_date add_weeks 1: " + (my_date add_weeks 1);
write "my_date add_days 1: " + (my_date add_days 1);
write "my_date add_hours 1: " + (my_date add_hours 1);
write "my_date add_minutes 1: " + (my_date add_minutes 1);
write "my_date add_seconds 1: " + (my_date add_seconds 1);

write "my_date subtract_years 1: " + (my_date subtract_years 1);
write "my_date subtract_months 1: " + (my_date subtract_months 1)
   ;
```

```
write "my_date subtract_weeks 1: " + (my_date subtract_weeks 1);
write "my_date subtract_days 1: " + (my_date subtract_days 1);
write "my_date subtract_hours 1: " + (my_date subtract_hours 1);
write "my_date subtract_minutes 1: " + (my_date subtract_minutes
   1);
write "my_date subtract_seconds 1: " + (my_date subtract_seconds
   1);
```

For the modelers, two global date variable are available: * `starting_date`: date considered as the beginning of the simulation * `current_date`: current date of the simulation

By default, these variables are nil. Defining a value of the starting_date allows to change the normal time management of the simulation by a more realistic one (using calendar):

```
global {
     date starting_date <- date([1979,12,17,19,45,10]);
}
```

When a variable is set to this variable, the current_date variable is automatically initialized with the same value. However, at each simulation step, the current_date variable is incremented by the step variable. The value of the current_date will replace the value of the time variable in the top left green panel.

Note that you have to be careful, when a real calendar is used, the built-in constants `#month` and `#year` should not be used as there are not consistent with the calendar (where month can be composed of 28, 29, 30 or 31 days).

Chapter 56

Implementing light

When using opengl display, GAMA provides you the possibility to manipulate one or several lights, making your display more realistic. Most of the following screenshots will be taken with the following short example gaml :

```
model test_light

grid cells {
    aspect base {
        draw square(1) at:{grid_x,grid_y} color:#white;
    }
}
experiment my_experiment type:gui{
    output {
        display my_display type:opengl background:#darkblue {
            species cells aspect:base;
            graphics "my_layer" {
                draw square(100) color:#white at:{50,50};
                draw cube(5) color:#lightgrey at:{50,30};
                draw cube(5) color:#lightgrey at:{30,35};
                draw cube(5) color:#lightgrey at:{60,35};
                draw sphere(5) color:#lightgrey at:{10,10,2.5};
                draw sphere(5) color:#lightgrey at:{20,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,60,2.5};
                draw cone3D(5,5) color:#lightgrey at:{55,10,0};
                draw cylinder(5,5) color:#lightgrey at:{10,60,0};
                }
            }
        }
    }
```

Index

Light generalities

Before going deep into the code, here is a quick explanation about how light works in opengl. First of all, you need to know that there are 3 types of lights you can manipulate : the **ambient light**, the **diffuse light** and the **specular light**. Each "light" in opengl is in fact composed of those 3 types of lights.

Ambient light

The **ambient light** is the light of your world without any lighting. If a face of a cube is not stricken by the light rays for instance, this face will appear totally black if there is no ambient light. To make your world more realistic, it is better to have an ambient light. An ambient light has then no position or direction. It is equally distributed to all the objects of your scene.

Here is an example of our GAML scene using only ambient light (color red) :

Diffuse light

The **diffuse light** can be seen as the light rays : if a face of a cube is stricken by the diffuse light, it will take the color of this diffuse light. You have to know that the more perpendicular the face of your object will be to the light ray, the more lightened the face will be.

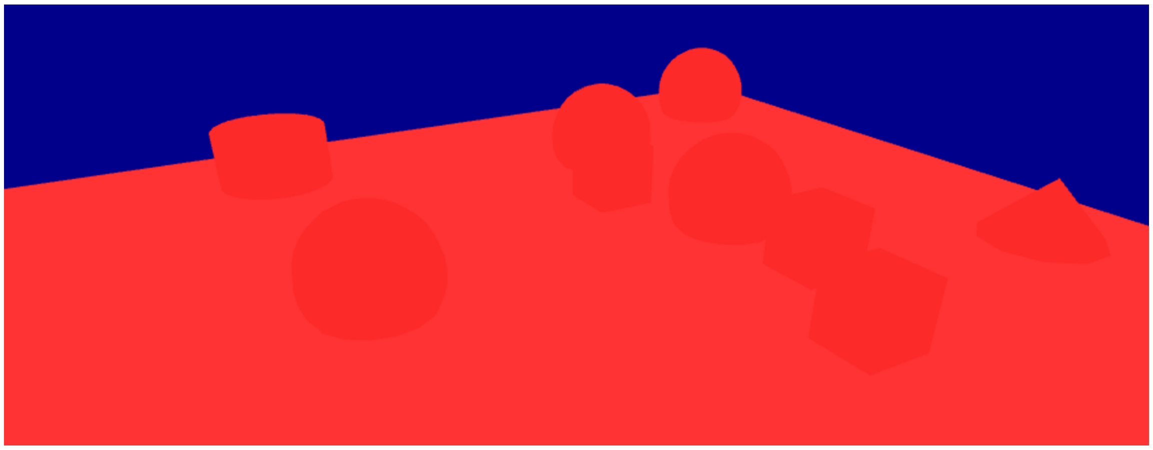


Figure 56.1: resources/images/lightRecipes/ambient_light.png

A diffuse light has then a direction. It can have also a position. Your have 2 categories of diffuse light : the **positional lights**, and the **directional lights**.

Positional lights

Those lights have a position in your world. It is the case of **point lights** and **spot lights**.

- Point lights

Points lights can be seen as a candle in your world, diffusing the light equally in all the direction.

Here is an example of our GAML scene using only diffuse light, with a point light (color red, the light source is displayed as a red sphere) :

- Spot lights

Spot lights can be seen as a torch light in your world. It needs a position, and also a direction and an angle.

Here is an example of our GAML scene using only diffusion light, with a spot light (color red, the light source is displayed as a red cone) :

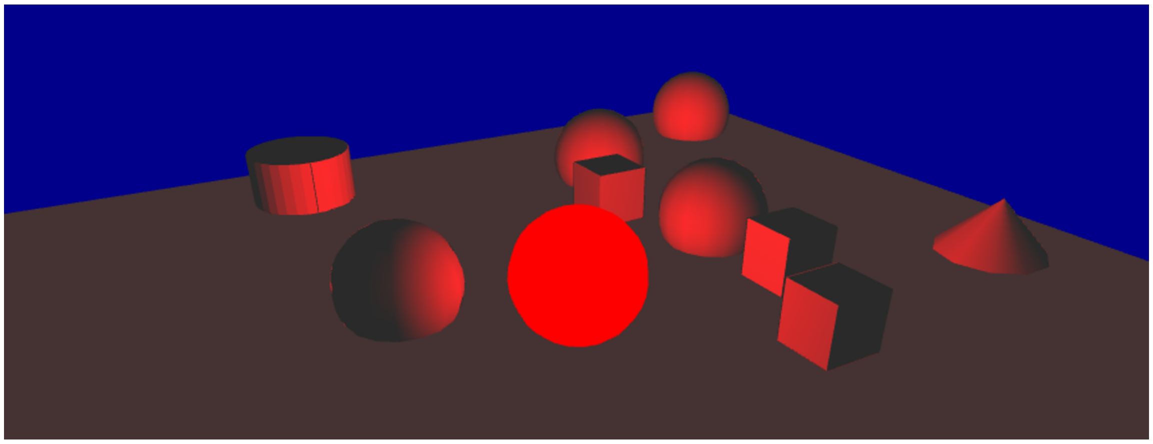


Figure 56.2: resources/images/lightRecipes/point_light.png

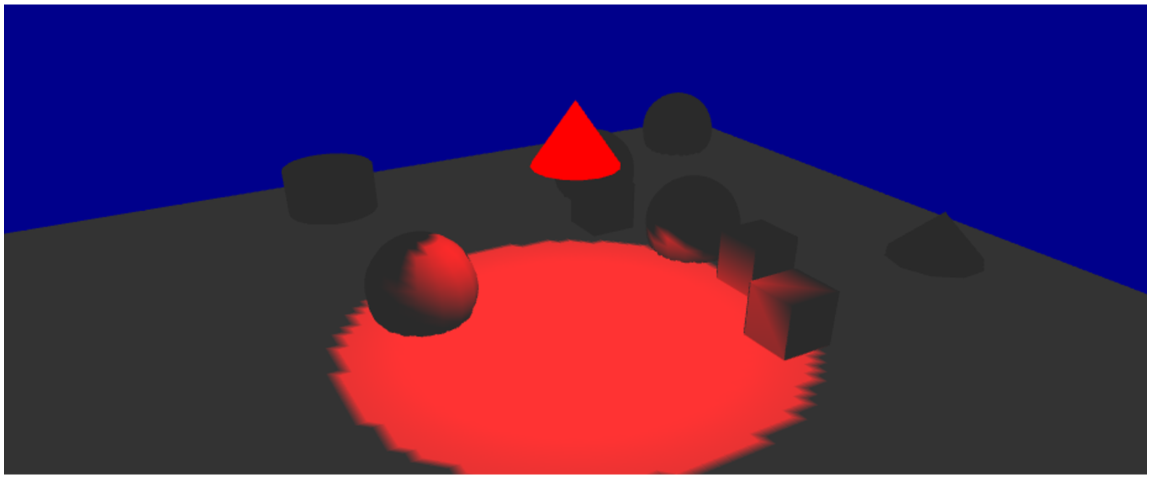


Figure 56.3: resources/images/lightRecipes/spot_light.png

Positional lights, as they have a position, can also have an attenuation according to the distance between the light source and the object. The value of positional lights are computed with the following formula : diffuse_light = diffuse_light * (1 / (1 + constante_attenuation + linear_attenuation * d + quadratic_attenuation * d)) By changing those 3 values (constante_attenuation, linear_attenuation and quadratic_attenuation), you can control the way light is diffused over your world (if your world is "foggy" for instance, you may turn your linear and quadratic attenuation on). Note that by default, all those attenuation are equal to 0.

Here is an example of our GAML scene using only diffusion light, with a point light with linear attenuation (color red, the light source is displayed as a red sphere) :

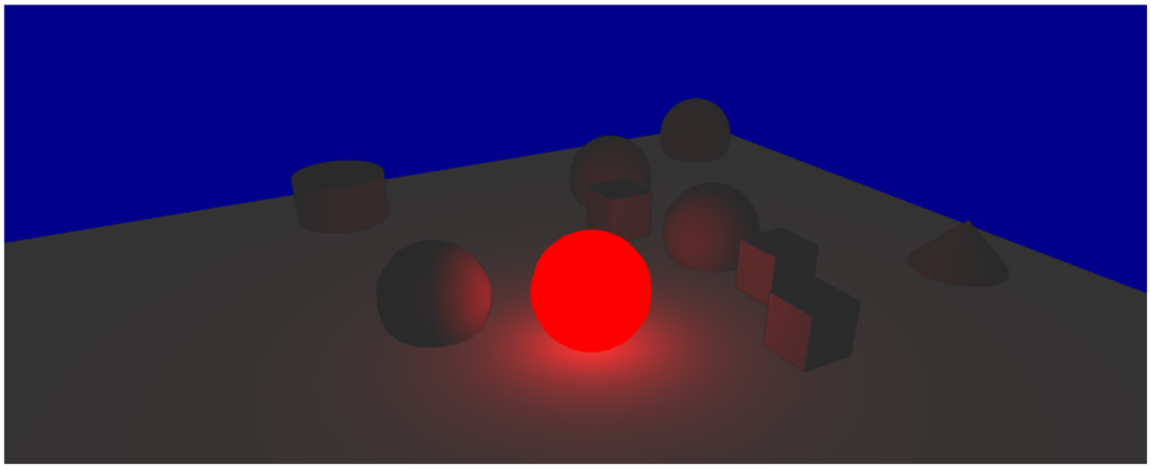


Figure 56.4: resources/images/lightRecipes/point_light_with_attenuation.png

Directional lights

Directional lights have no real "position" : they only have a direction. A directional light will strike all the objects of your world with the same direction. An example of directional light you have in the real world would be the light of the sun : the sun is so far away from us that you can consider that the rays have the same direction and the same intensity wherever they strike. Since there is no position for directional lights, there is no attenuation either.

Here is an example of our GAML scene using only diffusion light, with a directional light (color red) :

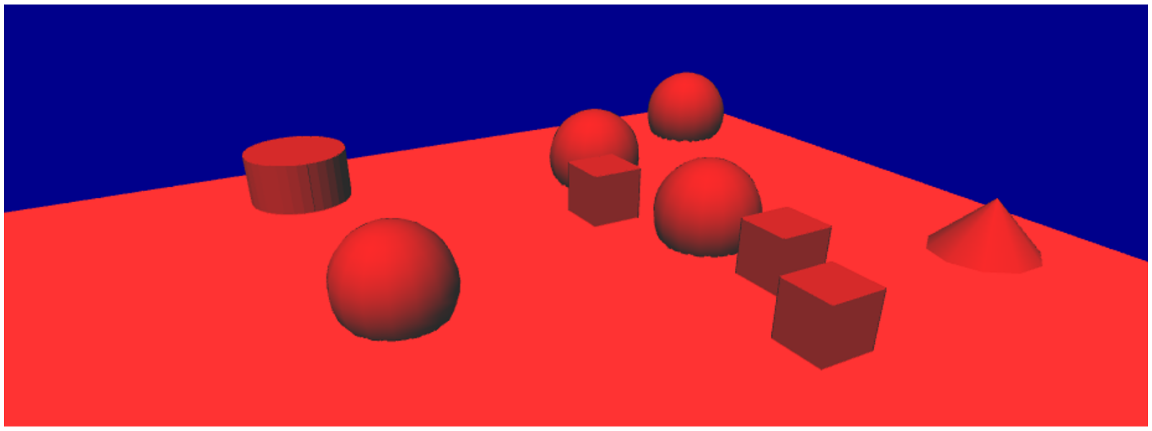


Figure 56.5: resources/images/lightRecipes/direction_light.png

Specular light

This is a more advanced concept, giving an aspect a little bit "shinny" to the objects stricken by the specular light. It is used to simulate the interaction between the light and a special material (ex : wood, steel, rubber…). This specular light is not implemented yet in gama, only the two others are.

Default light

In your opengl display, without specifying any light, you will have only one light, with those following properties :

Those values have been chosen in order to have the same visual effect in both opengl and java2D displays, when you display 2D objects, and also to have a nice "3D effect" when using the opengl displays. We chose the following setting by default : * The ambient light value : rgb(127,127,127,255) * diffuse light value : rgb(127,127,127,255) * type of light : direction * direction of the light : (0.5,0.5,-1);

Here is an example of our GAML scene using the default light :

Custom lights

In your opengl display, you can create up to 8 lights, giving them the properties you want.

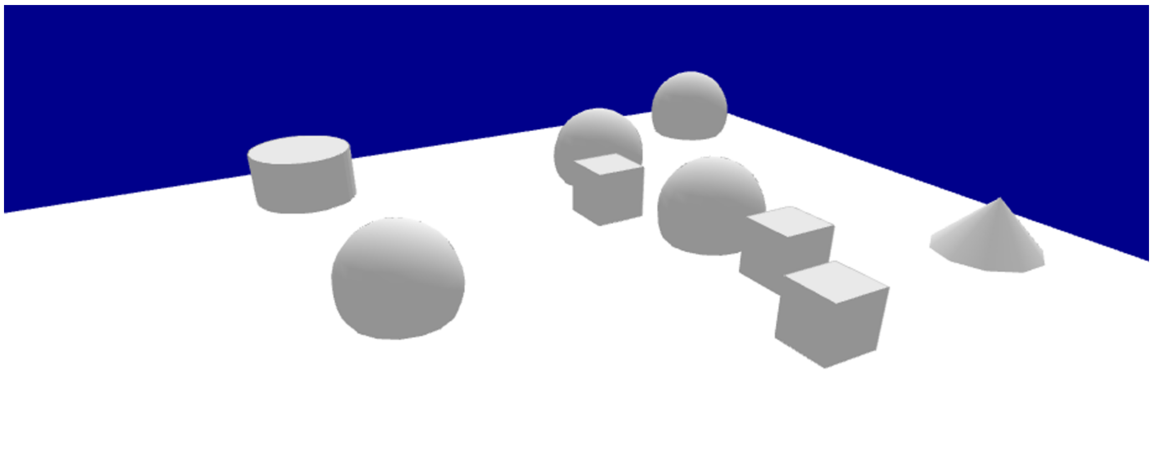


Figure 56.6: resources/images/lightRecipes/default_light.png

Ambient light

In order to keep it simple, the ambient light can be set directly when you are declaring your display, through the facet `ambient_light`. You will have one only ambient light.

```
experiment my_experiment type:gui {
    output {
        display "my_display" type:opengl ambient_light:100 {
        }
    }
}
```

Note for developers : Note that this ambient light is set to the GL_LIGHT0. This GL_-LIGHT0 only contains an ambient light, and no either diffuse nor specular light.

Diffuse light

In order to add lights, or modifying the existing lights, you can use the statement `light`, inside your `display` scope :

```
experiment my_experiment type:gui {
    output {
        display "my_display" type:opengl {
            light id:0;
        }
    }
}
```

This statement has just one non-optional facet : the facet "id". Through this facet, you can specify which light you want. You can control 7 lights, through an integer value between 1 and 7. Once you are manipulating a light through the `light` statement, the light is turned on. To switch off the light, you have to add the facet `active`, and turn it to `false`. The light you are declaring through the `light` statement is in fact a "diffuse" light. You can specify the color of the diffuse light through the facet `color` (by default, the color will be turn to white). An other very important facet is the `type` facet. This facet accepts a value among `direction`, `point` and `spot`.

Declaring direction light

A direction light, as explained in the first part, is a light without any position. Instead of the facet `position`, you will use the facet `direction`, giving a 3D vector.

Example of implementation :

```
light id:1 type:direction direction:{1,1,1} color:rgb(255,0,0);
```

Declaring point light

A point light will need a facet `position`, in order to give the position of the light source.

Example of implementation of a basic point light :

```
light id:1 type:point position:{10,20,10} color:rgb(255,0,0);
```

You can add, if you want, a custom attenuation of the light, through the facets `linear_attenuation` or `quadratic_attenuation`.

Example of implementation of a point light with attenuation :

```
light id:1 type:point position:{10,20,10} color:rgb(255,0,0)
   linear_attenuation:0.1;
```

Declaring spot light

A spot light will need the facet `position` (a spot light is a positionnal light) and the facet `direction`. A spot light will also need a special facet `spot_angle` to determine the angle of the spot (by default, this value is set to 45 degree).

Example of implementation of a basic spot light :

```
light id:1 type:spot position:{0,0,10} direction:{1,1,1} color:
    rgb(255,0,0) spot_angle:20;
```

Same as for point light, you can specify an attenuation for a spot light.

Example of implementation of a spot light with attenuation :

```
light id:1 type:spot position:{0,0,10} direction:{1,1,1} color:
    rgb(255,0,0) spot_angle:20;
```

Note that when you are working with lights, you can display your lights through the facet `draw light` to help you implementing your model. The three types of lights are displayed differently : - The **point** light is represented by a sphere with the color of the diffuse light you specified, in the position of your light source. - The **spot** light is represented by a cone with the color of the diffuse light you specified, in the position of your light source, the orientation of your light source. The size of the base of the cone will depend of the angle you specified. - The **direction** light, as it has no real position, is represented with arrows a bit above the world, with the direction of your direction light, and the color of the diffuse light you specified.

Note for developers : Note that, since the GL_LIGHT0 is already reserved for the ambient light (only !), all the other lights (from 1 to 7) are the lights from GL_LIGHT1 to GL_LIGHT7.

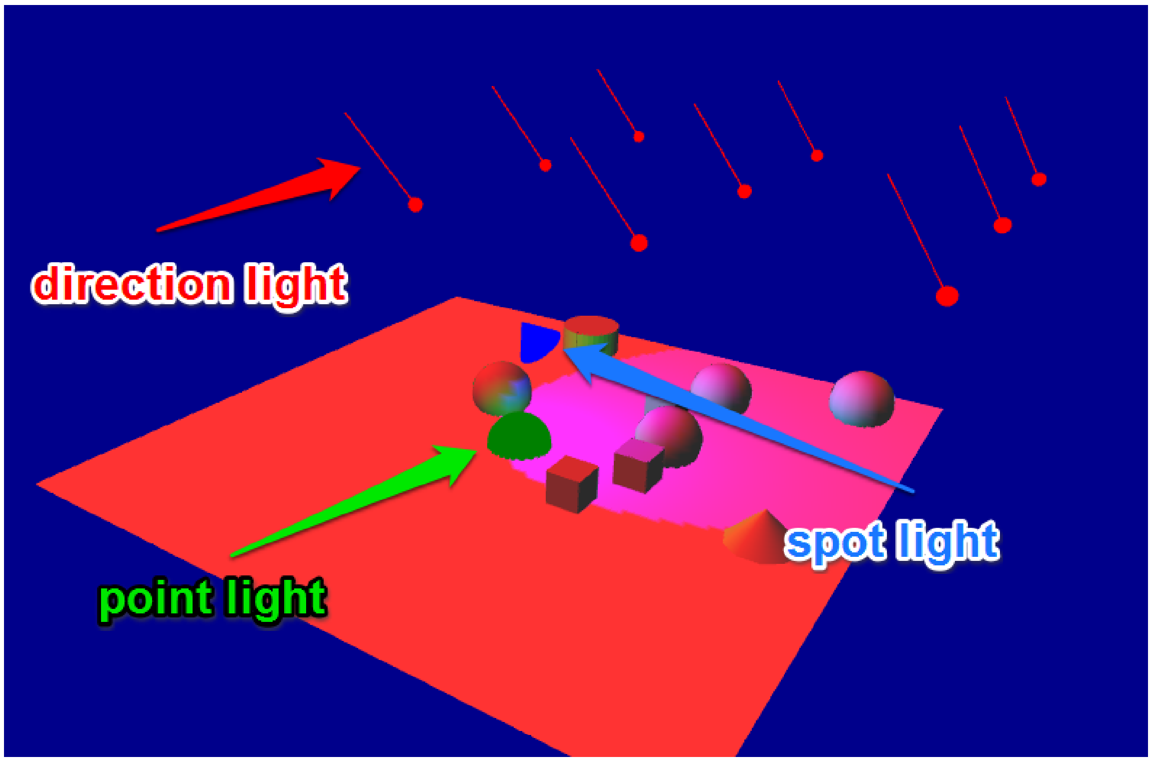


Figure 56.7: resources/images/lightRecipes/draw_light.png

Chapter 57

Using Comodel

Introduction

In the trend of developing complex system of multi-disciplinary, composing and coupling models are days by days become the most attractive research objectives. GAMA is supporting the co-modelling and co-simulation which are suppose to be the common coupling infrastructure.

Example of a Comodel

A Comodel is a model, especially an multi-agent-based, compose several sub-model, called micro-model. A comodel itself could be also a micro-model of an other comodel. From the view of a micro-model, comodel is called a macro-model.

A micro-model must be import, instantiate, and life-control by macro-model.

Why and when can we use Comodel ?

to be completed...

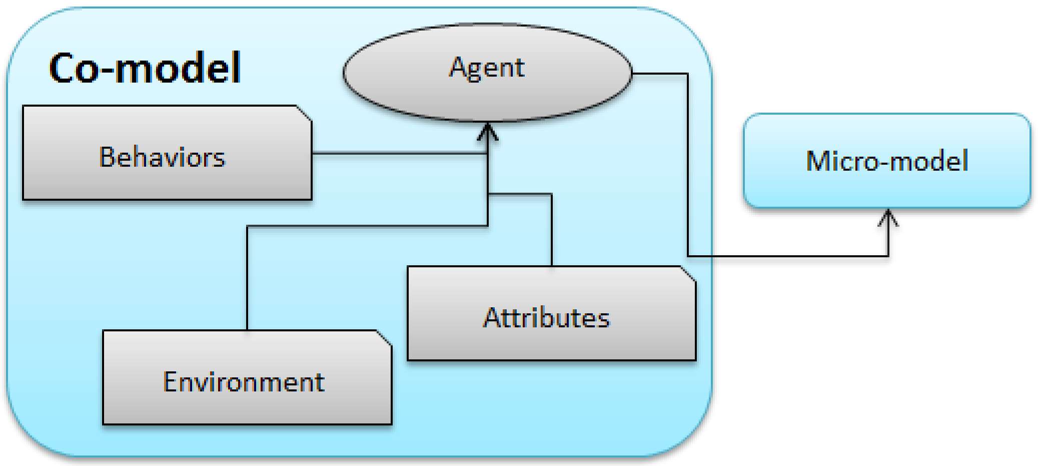


Figure 57.1:

Use of Comodel in a GAML model

The GAML language has been evolve by extend the import section. The old importation tell the compiler to merge all imported elements into as one model, but the new one allows modellers to keep the elements come from imported models separately with the caller model.

Defining a micro-model

Defining a micro-model of comodel is to import an existing model with an alias name. The syntax is:

```
import <path to the GAML model> as <identifier>
```

The identifier is then become the new name of the micro-model.

Instantiate a micro-model

After the importation and giving an identifier, micro-model must be explicitly instantiated. It could be done by create statement.

```
create <micro-model identifier> . <experiment name> [optional
   parameter];
```

THe is an expriment inside micro-model. This syntax will generate an experiment agent and attach an implicitly simulation.

Note: Creation of multi-instant is not create multi-simulation, but multi-experiment. Modellers could create a experiment with multi-simulation by explicitly do the init inside the experiment scope.

Control micro-model life-cycle

A micro-model can be control as the normal agent by asking the correspond identifier, and also be destroy by the 'o die' statement. As fact, it can be recreate any time we need.

```
ask (<micro-model identifier> . <experiment name>  at <number> )
   . simulation {
        ...
}
```

Visualize micro-model

The micro-model species could display in comodel with the support of agent layer

```
agents "name of layer" value: (<micro-model> . <experiment name>
   at <number>).<get List of agents>;
```

More details

Example of the comodel

Urbanization model with Traffic model

Flood model with Evacuation model

Reusing of two existing models:Flood Simulation and Evacuation.

Toy Models/Evacuation/models/continuous_move.gaml

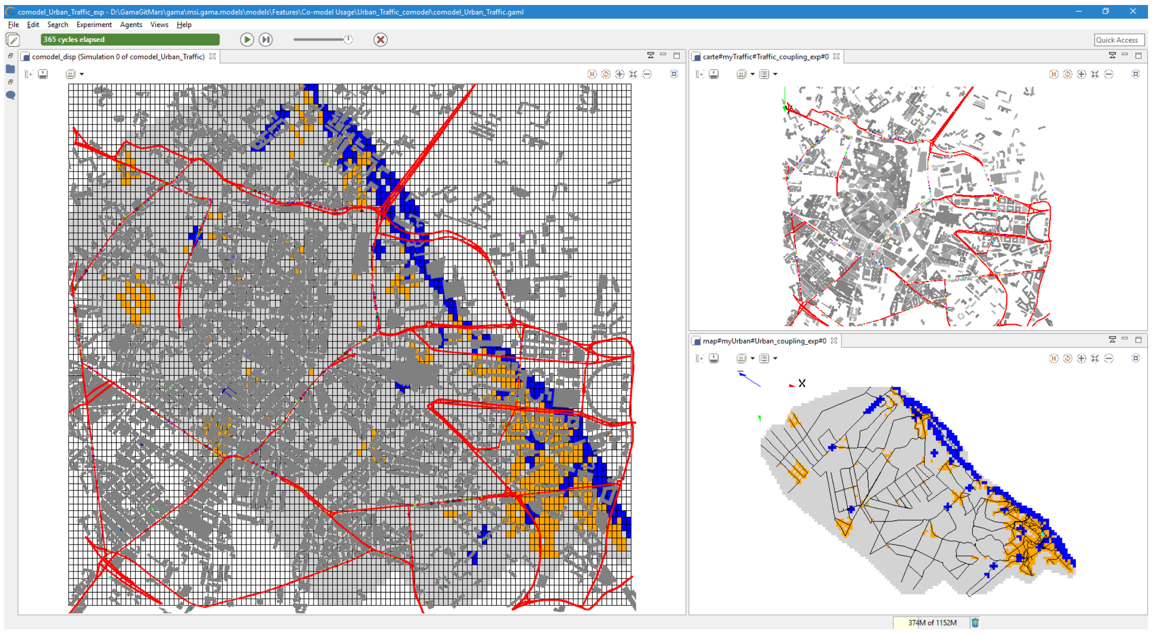

365 cycles elapsed
comodel_disp (Simulation 0 of comodel_Urban_Traffic)
carte#myTraffic#Traffic_coupling_exp#0
map#myUrban#Urban_coupling_exp#0
374M of 1152M

Figure 57.2:

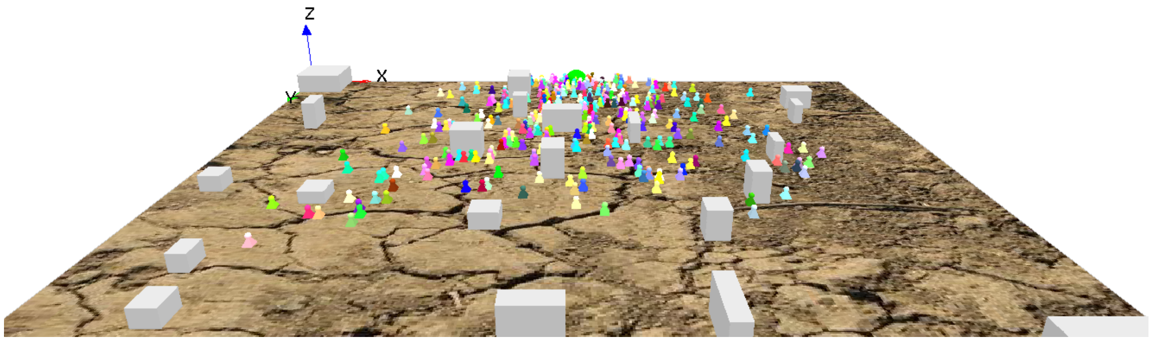


Figure 57.3:

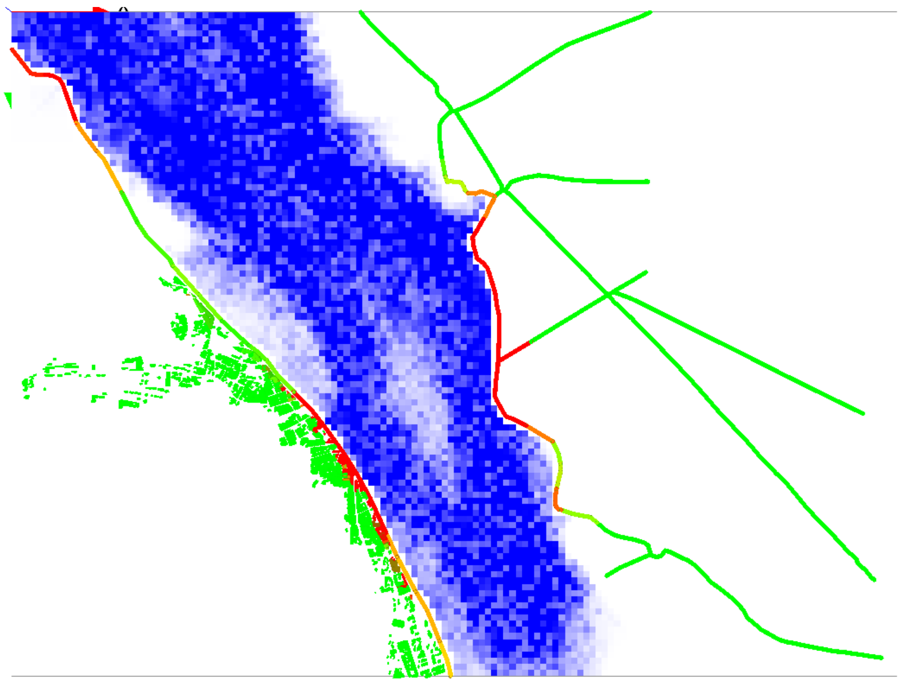


Figure 57.4:

Toy Models/Flood Simulation/models/Hydrological Model.gaml

The comodel explore the effect of flood on evacuation plan:

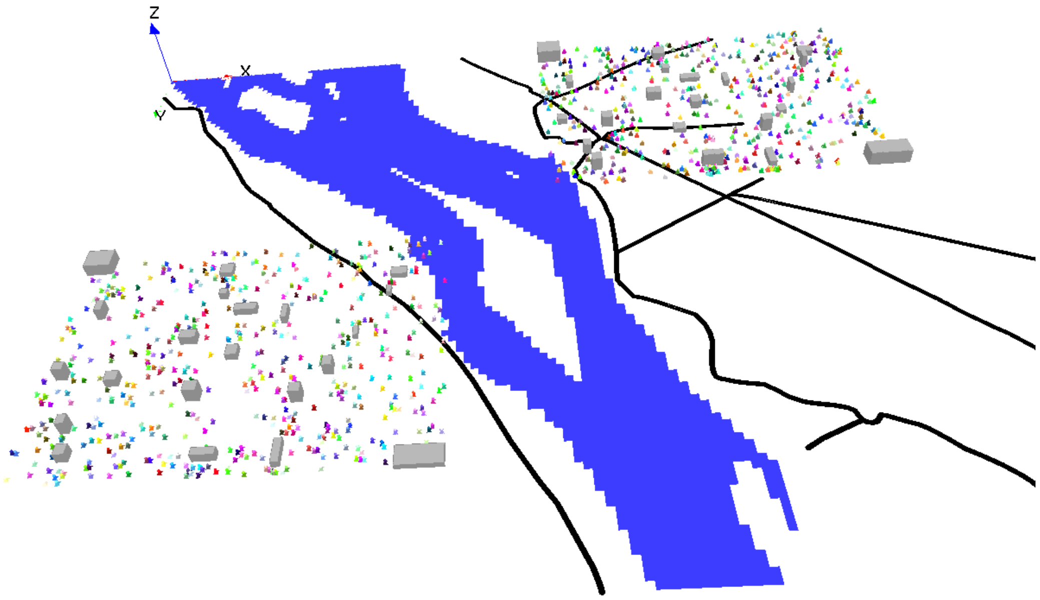


Figure 57.5:

Simulation results:

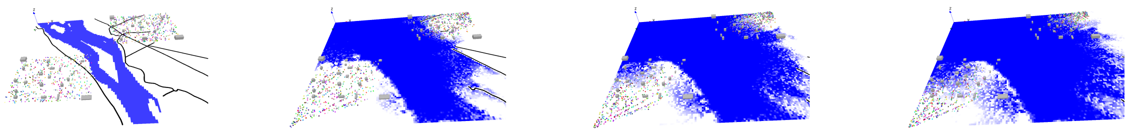


Figure 57.6:

Part III

GAML References (Documentation)

Built-in Species

This file is automatically generated from java files. Do Not Edit It.

It is possible to use in the models a set of built-in agents. These agents allow to directly use some advance features like clustering, multi-criteria analysis, etc. The creation of these agents are similar as for other kinds of agents:

```
create species: my_built_in_agent returns: the_agent;
```

So, for instance, to be able to use clustering techniques in the model:

```
create cluster_builder returns: clusterer;
```

Table of Contents

`agent`

Actions

`_init_`

- returns: `unknown`

`_step_`

- returns: `unknown`

`AgentDB`

Actions

`close`

- returns: `unknown`

`connect`

- returns: `unknown`
- ☐ **`params`** (`map`): Connection parameters

`executeUpdate`

- returns: `int`
- ☐ **`updateComm`** (`string`): SQL commands such as Create, Update, Delete, Drop with question mark
- ☐ **`values`** (`list`): List of values that are used to replace question mark

`getParameter`

- returns: `unknown`

`insert`

- returns: `int`
- ☐ `into` (`string`): Table name
- ☐ `columns` (`list`): List of column name of table
- ☐ `values` (`list`): List of values that are used to insert into table. Columns and values must have same size

`isConnected`

- returns: `bool`

`select`

- returns: `container`
- ☐ `select` (`string`): select string
- ☐ `values` (`list`): List of values that are used to replace question marks

`setParameter`

- returns: `unknown`
- ☐ `params` (`map`): Connection parameters

`testConnection`

- returns: `bool`
- ☐ `params` (`map`): Connection parameters

timeStamp

- returns: `float`

---

base_edge

Actions

---

experiment

Actions

---

graph_edge

Actions

---

graph_node

Actions

related_to

- returns: `bool`

- ☐ `other` (`agent`):

`model`

Actions

`halt`

Allows to stop the current simulation so that cannot be continued after. All the behaviors and updates are stopped. * returns: `unknown`

`pause`

Allows to pause the current simulation **ACTUALLY EXPERIMENT FOR THE MOMENT**. It can be set to continue with the manual intervention of the user. * returns: `unknown`

`physical_world`

Actions

`compute_forces`

- returns: `unknown`

Chapter 58

The 'agent' built-in species (Under Construction)

As described in the presentation of GAML, the hierarchy of species derives from a single built-in species called `agent`. All its components (attributes, actions) will then be inherited by all direct or indirect children species (including `model` and `experiment`), with the exception of species that explicitly mention `use_minimal_agents: true` as a facet, which inherit from a stripped-down version of `agent` (see below).

`agent` attributes

`agent` defines several attributes, which form the minimal set of knowledge any agent will have in a model. *

`agent` actions

Chapter 59

The 'model' built-in species (Under Construction)

As described in the presentation of GAML, any model in GAMA is a species (introduced by the keyword `global`) which directly inherits from an abstract species called `model`. This abstract species (sub-species of `agent`) defines several attributes and actions that can then be used in any global section of any model.

`model` attributes

`model` defines several attributes, which, in addition to the attributes inherited from `agent`, form the minimal set of knowledge a model can manipulate. *

`model` actions

Chapter 60

The 'experiment' built-in species (Under Construction)

As described in the presentation of GAML, any experiment attached to a model is a species (introduced by the keyword `experiment` which directly or indirectly inherits from an abstract species called `experiment` itself. This abstract species (sub-species of `agent`) defines several attributes and actions that can then be used in any experiment.

`experiment` attributes

`experiment` defines several attributes, which, in addition to the attributes inherited from `agent`, form the minimal set of knowledge any experiment will have access to.

`experiment` actions

Chapter 61

Built-in Skills

This file is automatically generated from java files. Do Not Edit It.

Introduction

Skills are built-in modules, written in Java, that provide a set of related built-in variables and built-in actions (in addition to those already provided by GAMA) to the species that declare them. A declaration of skill is done by filling the skills attribute in the species definition:

```
species my_species skills: [skill1, skill2] {
    ...
}
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. An example of skill is the `moving` skill.

So, for instance, if a species is declared as:

```
species foo skills: [moving]{
...
}
```

Its agents will automatically be provided with the following variables : `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `wander`, `follow` in addition to those built-in in species and declared by the modeller. Most of these variables, except the ones marked read-only, can be customized and modified like normal variables by the modeller. For instance, one could want to set a maximum for the speed; this would be done by redeclaring it like this:

```
float speed max:100 min:0;
```

Or, to obtain a speed increasing at each simulation step:

```
float speed max:100 min:0  <- 1 update: speed * 1.01;
```

Or, to change the speed in a behavior:

```
if speed = 5 {
    speed <- 10;
}
```

Table of Contents

---

advanced_driving

Variables

- `current_index` (`int`): the current index of the agent target (according to the targets list)

- **`current_lane`** (`int`): the current lane on which the agent is
- **`current_path`** (`path`): the current path that tha agent follow
- **`current_road`** (`agent`): current road on which the agent is
- **`current_target`** (`point`): the current target of the agent
- **`distance_to_goal`** (`float`): euclidean distance to the next point of the current segment
- **`final_target`** (`point`): the final target of the agent
- **`max_acceleration`** (`float`): maximum acceleration of the car for a cycle
- **`max_speed`** (`float`): maximal speed of the vehicle
- **`on_linked_road`** (`boolean`): is the agent on the linked road?
- **`proba_block_node`** (`float`): probability to block a node (do not let other driver cross the crossroad)
- **`proba_lane_change_down`** (`float`): probability to change lane to a lower lane (right lane if right side driving) if necessary
- **`proba_lane_change_up`** (`float`): probability to change lane to a upper lane (left lane if right side driving) if necessary
- **`proba_respect_priorities`** (`float`): probability to respect priority (right or left) laws
- **`proba_respect_stops`** (`list`): probability to respect stop laws - one value for each type of stop
- **`proba_use_linked_road`** (`float`): probability to change lane to a linked road lane if necessary
- **`real_speed`** (`float`): real speed of the agent (in meter/second)

- **`right_side_driving`** (`boolean`): are drivers driving on the right size of the road?
- **`security_distance_coeff`** (`float`): the coefficient for the computation of the the min distance between two drivers (according to the vehicle speed - security_distance = 1#m + security_distance_coeff ∗ real_speed)
- **`segment_index_on_road`** (`int`): current segment index of the agent on the current road
- **`speed`** (`float`): the speed of the agent (in meter/second)
- **`speed_coeff`** (`float`): speed coefficient for the speed that the driver want to reach (according to the max speed of the road)
- **`targets`** (`list`): the current list of points that the agent has to reach (path)
- **`vehicle_length`** (`float`): the length of the vehicle (in meters)

Actions

`advanced_follow_driving`

moves the agent towards along the path passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: float
- **`path`** (path): a path to be followed.
- **`target`** (point): the target to reach
- **`speed`** (float): the speed to use for this move (replaces the current value of speed)
- **`time`** (float): time to travel

`compute_path`

action to compute a path to a target location according to a given graph

- returns: path
- `graph` (graph): the graph on wich compute the path
- `target` (agent): the target node to reach
- `source` (agent): the source node (optional, if not defined, closest node to the agent location)
- `on_road` (agent): the road on which the agent is located (optional)

`drive`

action to drive toward the final target

- returns: void

`external_factor_impact`

action that allows to define how the remaining time is impacted by external factor

- returns: float
- `new_road` (agent): the road on which to the driver wants to go
- `remaining_time` (float): the remaining time

`is_ready_next_road`

action to test if the driver can take the given road at the given lane

- returns: bool

- `new_road` (agent): the road to test
- `lane` (int): the lane to test

`lane_choice`

action to choose a lane

- returns: int
- `new_road` (agent): the road on which to choose the lane

`path_from_nodes`

action to compute a path from a list of nodes according to a given graph

- returns: path
- `graph` (graph): the graph on wich compute the path
- `nodes` (list): the list of nodes composing the path

`speed_choice`

action to choose a speed

- returns: float
- `new_road` (agent): the road on which to choose the speed

`test_next_road`

action to test if the driver can take the given road

- returns: bool

- `new_road` (agent): the road to test

---

driving

Variables

- `lanes_attribute` (`string`): the name of the attribut of the road agent that determine the number of road lanes
- `living_space` (`float`): the min distance between the agent and an obstacle (in meter)
- `obstacle_species` (`list`): the list of species that are considered as obstacles
- `speed` (`float`): the speed of the agent (in meter/second)
- `tolerance` (`float`): the tolerance distance used for the computation (in meter)

Actions

`follow_driving`

moves the agent along a given path passed in the arguments while considering the other agents in the network.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `path` (path): a path to be followed.
- `return_path` (boolean): if true, return the path followed (by default: false)
- `move_weights` (map): Weigths used for the moving.

- `living_space` (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- `tolerance` (float): tolerance distance used for the computation (replaces the current value of tolerance)
- `lanes_attribute` (string): the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

`goto_driving`

moves the agent towards the target passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: path
- `target` (point,geometry,agent): the location or entity towards which to move.
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `on` (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)
- `return_path` (boolean): if true, return the path followed (by default: false)
- `move_weights` (map): Weigths used for the moving.
- `living_space` (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- `tolerance` (float): tolerance distance used for the computation (replaces the current value of tolerance)
- `lanes_attribute` (string): the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

---

fipa

The fipa skill offers some primitives and built-in variables which enable agent to communicate with each other using the FIPA interaction protocol.

Variables

- `accept_proposals` (`list`): A list of 'accept_proposal' performative messages of the agent's mailbox having .
- `agrees` (`list`): A list of 'accept_proposal' performative messages.
- `cancels` (`list`): A list of 'cancel' performative messages.
- `cfps` (`list`): A list of 'cfp' (call for proposal) performative messages.
- `conversations` (`list`): A list containing the current conversations of agent. Ended conversations are automatically removed from this list.
- `failures` (`list`): A list of 'failure' performative messages.
- `informs` (`list`): A list of 'inform' performative messages.
- `proposes` (`list`): A list of 'propose' performative messages .
- `queries` (`list`): A list of 'query' performative messages.
- `refuses` (`list`): A list of 'propose' performative messages.
- `reject_proposals` (`list`): A list of 'reject_proposals' performative messages.
- `requests` (`list`): A list of 'request' performative messages.
- `requestWhens` (`list`): A list of 'request-when' performative messages.
- `subscribes` (`list`): A list of 'subscribe' performative messages.

Actions

`accept_proposal`

Replies a message with an 'accept_proposal' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`agree`

Replies a message with an 'agree' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`cancel`

Replies a message with a 'cancel' peformative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`cfp`

Replies a message with a 'cfp' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`end_conversation`

Reply a message with an 'end_conversation' peprformative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsible of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`failure`

Replies a message with a 'failure' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`inform`

Replies a message with an 'inform' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`propose`

Replies a message with a 'propose' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`query`

Replies a message with a 'query' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`refuse`

Replies a message with a 'refuse' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The contents of the replying message

`reject_proposal`

Replies a message with a 'reject_proposal' performative message.

- returns: unknown

- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`reply`

Replies a message. This action should be only used to reply a message in a 'no-protocol' conversation and with a 'user defined performative'. For performatives supported by GAMA (i.e., standard FIPA performatives), please use the 'action' with the same name of 'performative'. For example, to reply a message with a 'request' performative message, the modeller should use the 'request' action.

- returns: unknown
- `message` (24): The message to be replied
- `performative` (string): The performative of the replying message
- `contents` (list): The content of the replying message

`request`

Replies a message with a 'request' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`send`

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.FIPAMessage

- `to` (list): A list of receiver agents
- `contents` (list): The content of the message. A list of any GAML type
- `performative` (string): A string, representing the message performative
- `protocol` (string): A string representing the name of interaction protocol

`start_conversation`

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.FIPAMessage
- `to` (list): A list of receiver agents
- `contents` (list): The content of the message. A list of any GAML type
- `performative` (string): A string, representing the message performative
- `protocol` (string): A string representing the name of interaction protocol

`subscribe`

Replies a message with a 'subscribe' performative message.

- returns: unknown
- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

GAMASQL

Variables

Actions

`read`

- returns: void
- `params` (map): Connection parameters
- `table` (string): select string with question marks
- `filter` (list): List of values that are used to replace question marks

`SqlObject`

- returns: msi.gama.database.geosql.GamaSqlConnection
- `params` (map): Connection parameters
- `table` (string): select string with question marks
- `filter` (string): Filter for select

`testConnection`

- returns: bool
- `params` (map): Connection parameters

grid

Variables

- `bands` (`list`): Represents the values of the different bands of the cell (list of floating point value automatically set when the grid is initialized from a grid file)
- `color` (`rgb`): Represents the color of the cell, used by default to represent the grid on displays
- `grid_value` (`float`): Represents a floating point value (automatically set when the grid is initialized from a grid file, and used by default to represent the elevation of the cell when displaying it on a display)
- `grid_x` (`int`): Returns the 0-based index of the column of the cell in the grid
- `grid_y` (`int`): Returns the 0-based index of the row of the cell in the grid
- `neighbors` (`list`): Represents the neighbor at distance 1 of the cell

Actions

MDXSKILL

Variables

Actions

`select`

- returns: list
- `params` (map): Connection parameters

- `onColumns` (string): select string with question marks
- `onRows` (list): List of values that are used to replace question marks
- `from` (list): List of values that are used to replace question marks
- `where` (list): List of values that are used to replace question marks
- `values` (list): List of values that are used to replace question marks

`testConnection`

- returns: bool
- `params` (map): Connection parameters

`timeStamp`

- returns: float

---

messaging

A simple skill that provides agents with a mailbox than can be filled with messages

Variables

- `mailbox` (`list`): The list of messages that can be consulted by the agent

Actions

`send`

- returns: msi.gama.extensions.messaging.GamaMessage

- `to` (any type): The agent, or server, to which this message will be sent to
- `contents` (any type): The contents of the message, an arbitrary object

---

moving

The moving skill is intended to define the minimal set of behaviours required for agents that are able to move on different topologies

Variables

- `destination` (`point`): Represents the next location of the agent if it keeps its current speed and heading (read-only)
- `heading` (`int`): Represents the absolute heading of the agent in degrees.
- `location` (`point`): Represents the current position of the agent
- `speed` (`float`): Represents the speed of the agent (in meter/second)

Actions

`follow`

moves the agent along a given path passed in the arguments.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `path` (path): a path to be followed.

- `move_weights` (map): Weights used for the moving.
- `return_path` (boolean): if true, return the path followed (by default: false)

`goto`

moves the agent towards the target passed in the arguments.

- returns: path
- `target` (agent,point,geometry): the location or entity towards which to move.
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `on` (graph): graph that restrains this move
- `recompute_path` (boolean): if false, the path is not recompute even if the graph is modified (by default: true)
- `return_path` (boolean): if true, return the path followed (by default: false)
- `move_weights` (map): Weights used for the moving.

`move`

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `heading` (int): the angle (in degree) of the target direction.
- `bounds` (geometry,agent): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

`wander`

Moves the agent towards a random location at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- returns: void
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `amplitude` (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- `bounds` (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

---

moving3D

The moving skill 3D is intended to define the minimal set of behaviours required for agents that are able to move on different topologies

Variables

- `destination` (`point`): continuously updated destination of the agent with respect to its speed and heading (read-only)
- `heading` (`int`): the absolute heading of the agent in degrees (in the range 0-359)
- `pitch` (`int`): the absolute pitch of the agent in degrees (in the range 0-359)
- `roll` (`int`): the absolute roll of the agent in degrees (in the range 0-359)
- `speed` (`float`): the speed of the agent (in meter/second)

Actions

`move`

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `heading` (int): int, optional, the direction to take for this move (replaces the current value of heading)
- `pitch` (int): int, optional, the direction to take for this move (replaces the current value of pitch)
- `heading` (int): int, optional, the direction to take for this move (replaces the current value of roll)
- `bounds` (geometry,agent): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

---

physics

Variables

- `collisionBound` (`map`):
- `density` (`float`):
- `mass` (`float`):
- `motor` (`point`):

- `space` (`agent`):
- `velocity` (`list`):

Actions

---

skill_road

Variables

- `agents_on` (`list`): for each lane of the road, the list of agents for each segment
- `all_agents` (`list`): the list of agents on the road
- `lanes` (`int`): the number of lanes
- `linked_road` (`-18`): the linked road: the lanes of this linked road will be usable by drivers on the road
- `maxspeed` (`float`): the maximal speed on the road
- `source_node` (`agent`): the source node of the road
- `target_node` (`agent`): the target node of the road

Actions

`register`

register the agent on the road at the given lane

- returns: void

- `agent` (agent): the agent to register on the road.
- `lane` (int): the lane index on which to register; if lane index >= number of lanes, then register on the linked road

`unregister`

unregister the agent on the road

- returns: void
- `agent` (agent): the agent to unregister on the road.

---

skill_road_node

Variables

- `block` (`map`): define the list of agents blocking the node, and for each agent, the list of concerned roads
- `priority_roads` (`list`): the list of priority roads
- `roads_in` (`list`): the list of input roads
- `roads_out` (`list`): the list of output roads
- `stop` (`list`): define for each type of stop, the list of concerned roads

Actions

SQLSKILL

Variables

Actions

`executeUpdate`

- returns: int
- `params` (map): Connection parameters
- `updateComm` (string): SQL commands such as Create, Update, Delete, Drop with question mark
- `values` (list): List of values that are used to replace question mark

`getCurrentDateTime`

- returns: string
- `dateFormat` (string): date format examples: 'yyyy-MM-dd' , 'yyyy-MM-dd HH:mm:ss'

`getDateOffset`

- returns: string
- `dateFormat` (string): date format examples: 'yyyy-MM-dd' , 'yyyy-MM-dd HH:mm:ss'
- `dateStr` (string): Start date
- `offset` (string): number on day to increase or decrease

`insert`

- returns: int
- `params` (map): Connection parameters
- `into` (string): Table name
- `columns` (list): List of column name of table
- `values` (list): List of values that are used to insert into table. Columns and values must have same size

`list2Matrix`

- returns: matrix
- `param` (list): Param: a list of records and metadata
- `getName` (boolean): getType: a boolean value, optional parameter
- `getType` (boolean): getType: a boolean value, optional parameter

`select`

- returns: container
- `params` (map): Connection parameters
- `select` (string): select string with question marks
- `values` (list): List of values that are used to replace question marks

`testConnection`

- returns: bool

- `params` (map): Connection parameters

`timeStamp`

- returns: float

Chapter 62

Built-in Architectures

This file is automatically generated from java files. Do Not Edit It.

INTRODUCTION

Table of Contents

fsm

Variables

- `state` (string): Returns the current state in which the agent is
- `states` (list): Returns the list of all possible states the agents can be in

Actions

---

probabilistic_tasks

Variables

Actions

---

reflex

Variables

Actions

---

simple_bdi

Variables

- `belief_base` (list):
- `charisma` (float):
- `current_plan` (any type):
- `desire_base` (list):
- `emotion_base` (list):
- `intention_base` (list):
- `intention_persistence` (float): intention persistence
- `plan_base` (list):
- `plan_persistence` (float): plan persistence
- `probabilistic_choice` (boolean):
- `receptivity` (float):
- `thinking` (list):
- `uncertainty_base` (list):
- `use_emotions_architecture` (boolean):

Actions

`add_belief`

add the predicate in the belief base. * returns: bool
* `predicate` (map): predicate to add as a belief

`add_desire`

adds the predicates is in the desire base. * returns: bool
* `predicate` (546704): predicate to add
* `todo` (546704): add the desire as a subintention of this parameter

`add_emotion`

add the emotion to the emotion base. * returns: bool
* `emotion` (546706): emotion to add to the base

`add_intention`

check if the predicates is in the desire base. * returns: bool
* `predicate` (map): predicate to check

`add_subintention`

adds the predicates is in the desire base. * returns: bool
* `predicate` (546704): predicate name
* `subintentions` (546704): the subintention to add to the predicate
* `add_as_desire` (boolean): add the subintention as a desire as well (by default, false)

`add_uncertainty`

add a predicate in the uncertainty base. * returns: bool
* `predicate` (map): predicate to check

`clear_beliefs`

clear the belief base * returns: bool

`clear_desires`

clear the desire base * returns: bool

`clear_intentions`

clear the intention base * returns: bool

`current_intention_on_hold`

puts the current intention on hold until the specified condition is reached or all subintentions are reached (not in desire base anymore). * returns: bool
* `until` (any type): the current intention is put on hold (fited plan are not considered) until specific condition is reached. Can be an expression (which will be tested), a list (of subintentions), or nil (by default the condition will be the current list of subintentions of the intention)

`get_belief`

get the predicate in the belief base (if several, returns the first one). * returns: predicate
* `predicate` (546704): predicate to get

`get_belief_with_name`

get the predicates is in the belief base (if several, returns the first one). * returns: predicate
* `name` (string): name of the predicate to check

`get_beliefs`

get the list of predicates is in the belief base * returns: msi.gama.util.IList
* `predicate` (546704): name of the predicates to check

get_beliefs_with_name

get the list of predicates is in the belief base with the given name. * returns: java.util.List
* `name` (string): name of the predicates to check

get_current_intention

returns the current intention (last entry of intention base). * returns: predicate

get_desire

get the predicates is in the desire base (if several, returns the first one). * returns: predicate
* `predicate` (546704): predicate to check

get_desire_with_name

get the predicates is in the belief base (if several, returns the first one). * returns: predicate
* `name` (string): name of the predicate to check

get_desires

get the list of predicates is in the belief base * returns: msi.gama.util.IList
* `predicate` (546704): name of the predicates to check

get_desires_with_name

get the list of predicates is in the belief base with the given name. * returns: java.util.List
* `name` (string): name of the predicates to check

get_emotion

get the emotion in the emotion base (if several, returns the first one). * returns: emotion
* `emotion` (546706): emotion to get

`get_intention`

get the predicates is in the belief base (if several, returns the first one). * returns: predicate
* `predicate` (546704): predicate to check

`get_intention_with_name`

get the predicates is in the belief base (if several, returns the first one). * returns: predicate
* `name` (string): name of the predicate to check

`get_intentions`

get the list of predicates is in the belief base * returns: msi.gama.util.IList
* `predicate` (546704): name of the predicates to check

`get_intentions_with_name`

get the list of predicates is in the belief base with the given name. * returns: java.util.List
* `name` (string): name of the predicates to check

`get_plans`

get the list of plans. * returns: java.util.List

`get_uncertainty`

get the predicates is in the uncertainty base (if several, returns the first one). * returns: predicate
* `predicate` (546704): predicate to check

`has_belief`

check if the predicates is in the belief base. * returns: bool
* `predicate` (546704): predicate to check

`has_desire`

check if the predicates is in the desire base. * returns: bool
* `predicate` (546704): predicate to check

`has_emotion`

check if the emotion is in the belief base. * returns: bool
* `emotion` (546706): emotion to check

`has_uncertainty`

check if the predicates is in the uncertainty base. * returns: bool
* `predicate` (546704): predicate to check

`is_current_intention`

check if the predicates is the current intention (last entry of intention base). * returns: bool
* `predicate` (546704): predicate to check

`remove_all_beliefs`

removes the predicates from the belief base. * returns: bool
* `predicate` (546704): predicate to remove

`remove_belief`

removes the predicate from the belief base. * returns: bool
* `predicate` (546704): predicate to remove

`remove_desire`

removes the predicates from the desire base. * returns: bool
* `predicate` (546704): predicate to add

`remove_emotion`

removes the emotion from the emotion base. * returns: bool
* `emotion` (546706): emotion to remove

`remove_intention`

removes the predicates from the desire base. * returns: bool
* `predicate` (546704): predicate to add
* `desire_also` (boolean): removes also desire

`remove_uncertainty`

removes the predicates from the desire base. * returns: bool
* `predicate` (546704): predicate to add

`replace_belief`

replace the old predicate by the new one. * returns: bool
* `old_predicate` (546704): predicate to remove
* `predicate` (546704): predicate to add

---

sorted_tasks

Variables

Actions

---

user_first

Variables

Actions

---

user_last

Variables

Actions

---

user_only

Variables

Actions

---

weighted_tasks

Variables

Actions

Chapter 63

Statements

This file is automatically generated from java files. Do Not Edit It.

Table of Contents

Statements by kinds

- **Batch method**

  - annealing, exhaustive, genetic, hill_climbing, reactive_tabu, save_batch, tabu,

- **Behavior**
  - aspect, plan, reflex, state, task, test, user_init, user_panel,

- **Experiment**
  - experiment,

- **Layer**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,

- **Output**
  - display, inspect, monitor, output, output_file, permanent,

- **Parameter**
  - parameter,

- **Sequence of statements or action**
  - action, ask, capture, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, switch, trace, transition, user_command, using,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, error, export, focus, focus_on, highlight, let, put, remove, return, rule, save, set, simulate, solve, status, unconscious_contagion, user_input, warn, write,

- **Species**

  - species,

- **Variable (container)**
  - Variable_container,

- **Variable (number)**
  - Variable_number,

- **Variable (regular)**
  - Variable_regular,

Statements by embedment

- **Behavior**
  - add, ask, capture, conscious_contagion, create, diffuse, do, emotional_contagion, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, solve, status, switch, trace, transition, unconscious_contagion, using, warn, write,
- **Environment**
  - species,
- **Experiment**
  - action, annealing, exhaustive, export, genetic, hill_climbing, output, parameter, permanent, reactive_tabu, reflex, save_batch, setup, simulate, state, tabu, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Layer**
  - draw, error, focus_on, highlight, if, let, loop, status, switch, trace, warn, write,
- **Model**

  - action, aspect, equation, experiment, output, perceive, plan, reflex, rule, run, setup, species, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,

- **Sequence of statements or action**
  - add, ask, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, solve, status, switch, trace, transition, unconscious_contagion, using, warn, write,
- **Single statement**
  - run,
- **Species**
  - action, aspect, equation, perceive, plan, reflex, rule, run, setup, simulate, species, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **action**
  - return,
- **aspect**
  - draw,
- **chart**
  - add, ask, data, datalist, do, put, remove, set, simulate, using,
- **display**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,
- **display_population**
  - display_population,
- **equation**
  - =,
- **fsm**

  - state, user_panel,
- **if**
  - else,
- **output**
  - display, inspect, monitor, output_file,
- **permanent**
  - display, inspect, monitor, output_file,
- **probabilistic_tasks**
  - task,
- **sorted_tasks**
  - task,
- **state**
  - enter, exit,
- **switch**
  - default, match,
- **test**
  - assert,
- **user_command**
  - user_input,
- **user_first**
  - user_panel,
- **user_init**
  - user_panel,
- **user_last**
  - user_panel,

- **user_only**
  - user_panel,
- **user_panel**
  - user_command,
- **weighted_tasks**
  - task,

General syntax

A statement represents either a declaration or an imperative command. It consists in a keyword, followed by specific facets, some of them mandatory (in bold), some of them optional. One of the facet names can be omitted (the one denoted as omissible). It has to be the first one.

```
statement_keyword expression1 facet2: expression2 ... ;
or
statement_keyword facet1: expression1 facet2: expression2 ...;
```

If the statement encloses other statements, it is called a **sequence statement**, and its sub-statements (either sequence statements or single statements) are declared between curly brackets, as in:

```
statement_keyword1 expression1 facet2: expression2... { // a
   sequence statement
     statement_keyword2 expression1 facet2: expression2...;  // a
    single statement
     statement_keyword3 expression1 facet2: expression2...;
}
```

=

Facets

- `right` (float), (omissible) : the right part of the equation (it is mandatory that it can be evaluated as a float

- `left` (any type): the left part of the equation (it should be a variable or a call to the diff() or diff2() operators)

Definition

Allows to implement an equation in the form function(n, t) = expression. The left function is only here as a placeholder for enabling a simpler syntax and grabbing the variable as its left member.

Usages

- The syntax of the = statement is a bit different from the other statements. It has to be used as follows (in an equation):

```
float t;
float S;
float I;
equation SI {
   diff(S,t) = (- 0.3 * S * I / 100);
   diff(I,t) = (0.3 * S * I / 100);
}
```

- See also: equation, solve,

Embedments

- The = statement is of type: **Single statement**
- The = statement can be embedded into: equation,
- The = statement embeds statements:

action

Facets

- `name` (an identifier), (omissible) : identifier of the action
- `index` (a datatype identifier): if the action returns a map, the type of its keys
- `of` (a datatype identifier): if the action returns a container, the type of its elements
- `type` (a datatype identifier): the action returned type
- `virtual` (boolean): whether the action is virtual (defined without a set of instructions) (false by default)

Definition

Allows to define in a species, model or experiment a new action that can be called elsewhere.

Usages

- The simplest syntax to define an action that does not take any parameter and does not return anything is:

```
action simple_action {
   // [set of statements]
}
```

- If the action needs some parameters, they can be specified betwee, braquets after the identifier of the action:

```
action action_parameters(int i, string s){
   // [set of statements using i and s]
}
```

- If the action returns any value, the returned type should be used instead of the "action" keyword. A return statement inside the body of the action statement is mandatory.

```
int action_return_val(int i, string s){
   // [set of statements using i and s]
   return i + i;
}
```

- If virtual: is true, then the action is abstract, which means that the action is defined without body. A species containing at least one abstract action is abstract. Agents of this species cannot be created. The common use of an abstract action is to define an action that can be used by all its sub-species, which should redefine all abstract actions and implements its body.

```
species parent_species {
   int virtual_action(int i, string s);
}

species children parent: parent_species {
   int virtual_action(int i, string s) {
      return i + i;
   }
}
```

- See also: do,

Embedments

- The `action` statement is of type: **Sequence of statements or action**
- The `action` statement can be embedded into: Species, Experiment, Model,
- The `action` statement embeds statements: return,

---

add

Facets

- `to` (any type in [container, species, agent, geometry]): an expression that evaluates to a container

- `item` (any type), (omissible) : any expression to add in the container
- `all` (any type): Allows to either pass a container so as to add all its element, or 'true', if the item to add is already a container.
- `at` (any type): position in the container of added element
- `edge` (any type): a pair that will be added to a graph as an edge (if nodes do not exist, they are also added)
- `node` (any type): an expression that will be added to a graph as a node.
- `vertex` (any type):
- `weight` (float):

Definition

Allows to add, i.e. to insert, a new element in a container (a list, matrix, map, …).Incorrect use: The addition of a new element at a position out of the bounds of the container will produce a warning and let the container unmodified. If all: is specified, it has no effect if its argument is not a container, or if its argument is 'true' and the item to add is not a container. In that latter case

Usages

- The new element can be added either at the end of the container or at a particular position.

```
add expr to: expr_container;     // Add at the end
add expr at: expr to: expr_container;   // Add at position expr
```

- Case of a list, the expression in the facet at: should be an integer.

```
list<int> workingList <- [];
add 0 at: 0 to: workingList ;   // workingList equals [0]
add 10 at: 0 to: workingList ;  // workingList equals [10,0]
add 20 at: 2 to: workingList ;  // workingList equals [10,0,20]
add 50 to: workingList;     // workingList equals [10,0,20,50]
add [60,70] all: true to: workingList;  // workingList equals
   [10,0,20,50,60,70]
```

- Case of a map: As a map is basically a list of pairs key::value, we can also use the add statement on it. It is important to note that the behavior of the statement is slightly different, in particular in the use of the at facet, which denotes the key of the pair.

```
map<string,string> workingMap <- [];
add "val1" at: "x" to: workingMap;  // workingMap equals ["x"::"
   val1"]
```

- If the at facet is omitted, a pair expr_item::expr_item will be added to the map. An important exception is the case where the expr_item is a pair: in this case the pair is added.

```
add "val2" to: workingMap;  // workingMap equals ["x"::"val1", "
   val2"::"val2"]
add "5"::"val4" to: workingMap;     // workingMap equals ["x"::"
   val1", "val2"::"val2", "5"::"val4"]
```

- Notice that, as the key should be unique, the addition of an item at an existing position (i.e. existing key) will only modify the value associated with the given key.

```
add "val3" at: "x" to: workingMap;  // workingMap equals ["x"::"
   val3", "val2"::"val2", "5"::"val4"]
```

- On a map, the all facet will add all value of a container in the map (so as pair val_-cont::val_cont)

```
add ["val4","val5"] all: true at: "x" to: workingMap;   //
   workingMap equals ["x"::"val3", "val2"::"val2", "5"::"val4","
   val4"::"val4","val5"::"val5"]
```

- In case of a graph, we can use the facets `node`, `edge` and `weight` to add a node, an edge or weights to the graph. However, these facets are now considered as deprecated, and it is advised to use the various edge(), node(), edges(), nodes() operators, which can build the correct objects to add to the graph

```
graph g <- as_edge_graph([{1,5}::{12,45}]);
add edge: {1,5}::{2,3} to: g;
list var <- g.vertices;     // var equals [{1,5},{12,45},{2,3}]
list var <- g.edges;    // var equals [polyline
   ({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]
add node: {5,5} to: g;
list var <- g.vertices;     // var equals
   [{1.0,5.0},{12.0,45.0},{2.0,3.0},{5.0,5.0}]
list var <- g.edges;    // var equals [polyline
   ({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]
```

- Case of a matrix: this statement can not be used on matrix. Please refer to the statement put.

- See also: put, remove,

Embedments

- The `add` statement is of type: **Single statement**
- The `add` statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The `add` statement embeds statements:

---

agents

Facets

- `value` (container): the set of agents to display
- `name` (a label), (omissible) : Human readable title of the layer
- `aspect` (an identifier): the name of the aspect that should be used to display the species
- `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false

- `focus` (agent): the agent on which the camera will be focused (it is dynamically computed)
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greter than 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, useful in case of agents that do not move)
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Definition

`agents` allows the modeler to display only the agents that fulfill a given condition.

Usages

- The general syntax is:

```
display my_display {
   agents layer_name value: expression [additional options];
}
```

- For instance, in a segregation model, `agents` will only display unhappy agents:

```
display Segregation {
   agents agentDisappear value: people as list where (each.
   is_happy = false) aspect: with_group_color;
}
```

- See also: display, chart, event, graphics, display_grid, image, overlay, display_population,

Embedments

- The `agents` statement is of type: **Layer**
- The `agents` statement can be embedded into: display,
- The `agents` statement embeds statements:

---

annealing

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `nb_iter_cst_temp` (int): number of iterations per level of temperature
- `temp_decrease` (float): temperature decrease coefficient
- `temp_end` (float): final temperature
- `temp_init` (float): initial temperature

Definition

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the annealing statement uses `method annealing` instead of the expected `annealing name: id` :

```
method annealing [facet: value];
```

- For example:

```
method annealing temp_init: 100  temp_end: 1 temp_decrease: 0.5
   nb_iter_cst_temp: 5 maximize: food_gathered;
```

Embedments

- The `annealing` statement is of type: **Batch method**
- The `annealing` statement can be embedded into: Experiment,
- The `annealing` statement embeds statements:

---

ask

Facets

- `target` (any type in [container, agent]), (omissible) : an expression that evaluates to an agent or a list of agents
- `as` (species): an expression that evaluates to a species

Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. If the value of the target facet is nil or empty, the statement is ignored.

Usages

- Ask a set of receiver agents, stored in a container, to perform a block of statements. The block is evaluated in the context of the agents' species

```
ask ${receiver_agents} {
     ${cursor}
}
```

- Ask one agent to perform a block of statements. The block is evaluated in the context of the agent's species

```
ask ${one_agent} {
     ${cursor}
}
```

- If the species of the receiver agent(s) cannot be determined, it is possible to force it using the `as` facet. An error is thrown if an agent is not a direct or undirect instance of this species

```
ask${receiver_agent(s)} as: ${a_species_expression} {
     ${cursor}
}
```

- To ask a set of agents to do something only if they belong to a given species, the `of_species` operator can be used. If none of the agents belong to the species, nothing happens

```
ask ${receiver_agents} of_species ${species_name} {
     ${cursor}
}
```

- Any statement can be declared in the block statements. All the statements will be evaluated in the context of the receiver agent(s), as if they were defined in their species, which means that an expression like `self` will represent the receiver agent and not the sender. If the sender needs to refer to itself, some of its own attributes (or temporary variables) within the block statements, it has to use the keyword `myself`.

```
species animal {
    float energy <- rnd (1000) min: 0.0 {
    reflex when: energy > 500 { // executed when the energy is
   above the given threshold
         list<animal> others <- (animal at_distance 5); // find
   all the neighboring animals in a radius of 5 meters
         float shared_energy  <- (energy - 500) / length (others)
   ; // compute the amount of energy to share with each of them
         ask others { // no need to cast, since others has
   already been filtered to only include animals
              if (energy < 500) { // refers to the energy of each
    animal in others
                   energy <- energy + myself.shared_energy; //
   increases the energy of each animal
                   myself.energy <- myself.energy - myself.
   shared_energy; // decreases the energy of the sender
              }
         }
    }
}
```

- If the species of the receiver agent cannot be determined, it is possible to force it by casting the agent. Nothing happens if the agent cannot be casted to this species

Embedments

- The `ask` statement is of type: **Sequence of statements or action**
- The `ask` statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The `ask` statement embeds statements:

aspect

Facets

- `name` (an identifier), (omissible) : identifier of the aspect (it can be used in a display to identify which aspect should be used for the given species). Two special names can also be used: 'default' will allow this aspect to be used as a replacement for the default aspect defined in preferences; 'highlighted' will allow the aspect to be used when the agent is highlighted as a replacement for the default (application of a color)

Definition

Aspect statement is used to define a way to draw the current agent. Several aspects can be defined in one species. It can use attributes to customize each agent's aspect. The aspect is evaluate for each agent each time it has to be displayed.

Usages

- An example of use of the aspect statement:

```
species one_species {
    int a <- rnd(10);
    aspect aspect1 {
        if(a mod 2 = 0) { draw circle(a);}
        else {draw square(a);}
        draw text: "a= " + a color: #black size: 5;
    }
}
```

Embedments

- The `aspect` statement is of type: **Behavior**
- The `aspect` statement can be embedded into: Species, Model,
- The `aspect` statement embeds statements: draw,

assert

Facets

- value (any type), (omissible) : the value that is evaluated and compared to other facets
- equals (any type): an expression, assert tests whether the value is equals to this expression
- is_not (any type): an expression, assert tests whether the value is not equals to this expression
- raises (an identifier): “error” or “warning”, used in testing what raises the evaluation of the value: expression

Definition

Allows to check whether the evaluation of a given expression fulfills a given condition. If it is not fulfilled, an exception is raised.

Usages

- if the equals: facet is used, the equality between the evaluation of expressions in the value: and in the equals: facets is tested

```
assert (2+2) equals: 4;
```

- if the is_not: facet is used, the inequality between the evaluation of expressions in the value: and in the equals: facets is tested

```
assert self is_not: nil;
```

- if the raises: facet is used with either “warning” or “error”, the statement tests whether the evaluation of the value: expression raises an error (resp. a warning)

```
int z <- 0;
assert (3/z) raises: "error";
```

- See also: test, setup,

Embedments

- The `assert` statement is of type: **Single statement**
- The `assert` statement can be embedded into: test,
- The `assert` statement embeds statements:

---

break

Facets

Definition

`break` allows to interrupt the current sequence of statements.

Usages

Embedments

- The `break` statement is of type: **Single statement**
- The `break` statement can be embedded into: Sequence of statements or action,
- The `break` statement embeds statements:

---

camera

Facets

- `name` (string), (omissible) : The name of the camera
- `location` (point): The location of the camera in the world
- `look_at` (point): The location that the camera is looking
- `up_vector` (point): The up-vector of the camera.

Definition

`camera` allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way.

Usages

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

Embedments

- The `camera` statement is of type: **Layer**
- The `camera` statement can be embedded into: display,
- The `camera` statement embeds statements:

---

capture

Facets

- `target` (any type in [agent, container]), (omissible) : an expression that is evaluated as an agent or a list of the agent to be captured
- `as` (species): the species that the captured agent(s) will become, this is a micro-species of the calling agent's species
- `returns` (a new identifier): a list of the newly captured agent(s)

Definition

Allows an agent to capture other agent(s) as its micro-agent(s).

Usages

- The preliminary for an agent A to capture an agent B as its micro-agent is that the A's species must defined a micro-species which is a sub-species of B's species (cf. [Species161#Nesting_species Nesting species]).

```
species A {
...
}
species B {
...
   species C parent: A {
   ...
   }
...
}
```

- To capture all "A" agents as "C" agents, we can ask an "B" agent to execute the following statement:

```
capture list(B) as: C;
```

- Deprecated writing:

```
capture target: list (B) as: C;
```

- See also: release,

Embedments

- The `capture` statement is of type: **Sequence of statements or action**
- The `capture` statement can be embedded into: Behavior, Sequence of statements or action,
- The `capture` statement embeds statements:

chart

Facets

- `name` (a label), (omissible) : the identifier of the chart layer
- `axes` (rgb): the axis color
- `background` (rgb): the background color
- `color` (rgb): Text color
- `gap` (float): minimum gap between bars (in proportion)
- `label_font` (string): Label font face
- `label_font_size` (int): Label font size
- `label_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display labels
- `legend_font` (string): Legend font face
- `legend_font_size` (int): Legend font size
- `legend_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display legend
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greter than 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `reverse_axes` (boolean): reverse X and Y axis (for example to get horizental bar charts
- `series_label_position` (an identifier), takes values in: {default, none, legend, onchart, yaxis, xaxis}: Position of the Series names: default (best guess), none, legend, onchart, xaxis (for category plots) or yaxis (uses the first serie name).
- `size` (point): the layer resize factor: {1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer. In case of a 3D layer, a 3D point can be used (note that {1,1} is equivalent to {1,1,0}, so a resize of a layer containing 3D objects with a 2D points will remove the elevation)
- `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded, default}: The sub-style style, also default style for the series.
- `tick_font` (string): Tick font face
- `tick_font_size` (int): Tick font size
- `tick_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display ticks
- `title_font` (string): Title font face

- `title_font_size` (int): Title font size
- `title_font_style` (an identifier), takes values in: {plain, bold, italic}: the style used to display titles
- `type` (an identifier), takes values in: {xy, scatter, histogram, series, pie, radar, heatmap, box_whisker}: the type of chart. It could be histogram, series, xy, pie, radar, heatmap or box whisker. The difference between series and xy is that the former adds an implicit x-axis that refers to the numbers of cycles, while the latter considers the first declaration of data to be its x-axis.
- `x_label` (a label): the title for the X axis
- `x_range` (any type in [float, int, point, list]): range of the x-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `x_serie` (any type in [list, float, int]): for series charts, change the default common x serie (simulation cycle) for an other value (list or numerical).
- `x_serie_labels` (any type in [list, float, int, a label]): change the default common x series labels (replace x value or categories) for an other value (string or numerical).
- `x_tick_unit` (float): the tick unit for the y-axis (distance between horyzontal lines and values on the left of the axis).
- `y_label` (a label): the title for the Y axis
- `y_range` (any type in [float, int, point, list]): range of the y-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `y_serie_labels` (any type in [list, float, int, a label]): for heatmaps/3d charts, change the default y serie for an other value (string or numerical in a list or cumulative).
- `y_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).

Definition

`chart` allows modeler to display a chart: this enables to display specific values of the model at each iteration. GAMA can display various chart types: time series (series), pie charts (pie) and histograms (histogram).

Usages

- The general syntax is:

```
display chart_display {
   chart "chart name" type: series [additional options] {
      [Set of data, datalists statements]
   }
}
```

- See also: display, agents, event, graphics, display_grid, image, overlay, quadtree, display_population, text,

Embedments

- The `chart` statement is of type: **Layer**
- The `chart` statement can be embedded into: display,
- The `chart` statement embeds statements: add, ask, data, datalist, do, put, remove, set, simulate, using,

---

conscious_contagion

Facets

- `emotion_created` (546706): the emotion that will be created with the contagion

- `emotion_detected` (546706): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly add an emotion of a perceived specie if the perceived agent ges a patricular emotion.

Usages

- Other examples of use:

```
conscious_contagion emotion_detected:fear emotion_created:
   fearConfirmed;
conscious_contagion emotion_detected:fear emotion_created:
   fearConfirmed charisma: 0.5 receptivity: 0.5;
```

Embedments

- The `conscious_contagion` statement is of type: **Single statement**
- The `conscious_contagion` statement can be embedded into: Behavior, Sequence of statements or action,
- The `conscious_contagion` statement embeds statements:

---

create

Facets

- `species` (any type in [species, agent]), (omissible) : an expression that evaluates to a species, the species of the agents to be created. In the case of simulations, the name 'simulation', which represents the current instance of simulation, can also be used as a proxy to their species
- `as` (species):
- `from` (any type): an expression that evaluates to a localized entity, a list of localized entities, a string (the path of a file), a file (shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a database
- `header` (boolean): an expression that evaluates to a boolean, when creating agents from csv file, specify whether the file header is loaded
- `number` (int): an expression that evaluates to an int, the number of created agents
- `returns` (a new identifier): a new temporary variable name containing the list of created agents (a list, even if only one agent has been created)
- `with` (map): an expression that evaluates to a map, for each pair the key is a species attribute and the value the assigned value

Definition

Allows an agent to create `number` agents of species `species`, to create agents of species `species` from a shapefile or to create agents of species `species` from one or several localized entities (discretization of the localized entity geometries).

Usages

- Its simple syntax to create `an_int` agents of species `a_species` is:

```
create a_species number: an_int;
create species_of(self) number: 5 returns: list5Agents;
5
```

- In GAML modelers can create agents of species `a_species  (with two attributes` type`and`nature`with types corresponding to the types of the shapefile attributes) from a shapefile`the_shapefile‘ while reading attributes ‘TYPE_-OCC’ and ‘NATURE’ of the shapefile. One agent will be created by object contained in the shapefile:

```
create a_species from: the_shapefile with: [type:: 'TYPE_OCC',
   nature::'NATURE'];
```

- In order to create agents from a .csv file, facet `header` can be used to specified whether we can use columns header:

```
create toto from: "toto.csv" header: true with:[att1::read("NAME"
   ), att2::read("TYPE")];
or
create toto from: "toto.csv" with:[att1::read(0), att2::read(1)];
    //with read(int), the index of the column
```

- Similarly to the creation from shapefile, modelers can create agents from a set of geometries. In this case, one agent per geometry will be created (with the geometry as shape)

```
create species_of(self) from: [square(4),circle(4)];    // 2
   agents have been created, with shapes respectively square(4)
   and circle(4)
```

- Created agents are initialized following the rules of their species. If one wants to refer to them after the statement is executed, the returns keyword has to be defined: the agents created will then be referred to by the temporary variable it declares. For instance, the following statement creates 0 to 4 agents of the same species as the sender, and puts them in the temporary variable children for later use.

```
create species (self) number: rnd (4) returns: children;
ask children {
        // ...
}
```

- If one wants to specify a special initialization sequence for the agents created, create provides the same possibilities as ask. This extended syntax is:

```
create a_species number: an_int {
     [statements]
}
```

- The same rules as in ask apply. The only difference is that, for the agents created, the assignments of variables will bypass the initialization defined in species. For instance:

```
create species(self) number: rnd (4) returns: children {
     set location <- myself.location + {rnd (2), rnd (2)}; //
   tells the children to be initially located close to me
     set parent <- myself; // tells the children that their
   parent is me (provided the variable parent is declared in this
    species)
}
```

- Despprecated uses:

```
// Simple syntax
create species: a_species number: an_int;
```

- If `number` equals 0 or species is not a species, the statement is ignored.

Embedments

- The `create` statement is of type: **Sequence of statements or action**
- The `create` statement can be embedded into: Behavior, Sequence of statements or action,
- The `create` statement embeds statements:

---

data

Facets

- `legend` (string), (omissible) :

- `value` (any type in [float, point, list]):
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (any type in [rgb, list]): color of the serie, for heatmap can be a list to specify [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false)
- `line_visible` (boolean): Line visible or not
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_-rectangle, marker_left_triangle}: Shape of the marker
- `marker_size` (list): Size of the marker. Can be a double (same size for every marker) or a list (different sizes for each marker.

- `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified on chart statement)
- `x_err_values` (any type in [float, list]): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_err_values` (any type in [float, list]): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

Embedments

- The `data` statement is of type: **Single statement**
- The `data` statement can be embedded into: chart, Sequence of statements or action,
- The `data` statement embeds statements:

---

datalist

Facets

- `value` (list): the values to display. Has to be a matrix, a list or a List of List. Each element can be a number (series/histogram) or a list with two values (XY chart)
- `legend` (list), (omissible) : the name of the series: a list of strings (can be a variable with dynamic names)
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (list): list of colors, for heatmaps can be a list of [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false), same for all series.
- `line_visible` (boolean): Line visible or not (same for all series)
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_-rectangle, marker_left_triangle}: Shape of the marker. Same one for all series.

- `marker_size` (list): the marker sizes to display. Can be a list of numbers (same size for each marker of the series) or a list of list (different sizes by point)
- `style` (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified on chart statement)
- `x_err_values` (list): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_err_values` (list): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

Embedments

- The `datalist` statement is of type: **Single statement**
- The `datalist` statement can be embedded into: chart, Sequence of statements or action,
- The `datalist` statement embeds statements:

---

default

Facets

- `value` (any type), (omissible) :

Definition

Used in a switch match structure, the block prefixed by default is executed only if no other block has matched (otherwise it is not).

Usages

- See also: switch, match,

Embedments

- The `default` statement is of type: **Sequence of statements or action**
- The `default` statement can be embedded into: switch,
- The `default` statement embeds statements:

---

diffuse

Facets

- `var` (an identifier), (omissible) : the variable to be diffused

- `on` (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the proportion value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix masking the diffusion (matrix created from a image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- `mat_diffu` (matrix): the diffusion matrix (can have any size)
- `matrix` (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- `min_value` (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- `propagation` (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : S' = (S / N

/ proportion) - variation. The intensity of S is then diminished by S * proportion on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals 'gradient', the original intensity is not modified, and each neighbors receives the intensity : S / proportion - variation. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to 'diffusion'.
- `proportion` (float): a diffusion rate
- `radius` (int): a diffusion radius (in number of cells from the center)
- `variation` (float): an absolute value to decrease at each neighbors

Definition

This statements allows a value to diffuse among a species on agents (generally on a grid) depending on a given diffusion matrix.

Usages

- A basic example of diffusion of the variable phero defined in the species cells, given a diffusion matrix math_diff is:

```
matrix<float> math_diff <- matrix
   ([[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]);
diffuse var: phero on: cells mat_diffu: math_diff;
```

- The diffusion can be masked by obstacles, created from a bitmap image:

```
diffuse var: phero on: cells mat_diffu: math_diff mask: mymask;
```

- A convenient way to have an uniform diffusion in a given radius is (which is equivalent to the above diffusion):

```
diffuse var: phero on: cells proportion: 1/9 radius: 1;
```

Embedments

- The `diffuse` statement is of type: **Single statement**
- The `diffuse` statement can be embedded into: Behavior, Sequence of statements or action,
- The `diffuse` statement embeds statements:

---

display

Facets

- `name` (a label), (omissible) : the identifier of the display
- `ambient_light` (any type in [int, rgb]): Allows to define the value of the ambient light either using an int (ambient_light:(125)) or a rgb color ((ambient_-light:rgb(255,255,255)). default is rgb(127,127,127,255)
- `autosave` (any type in [boolean, point]): Allows to save this display on disk. A value of true/false will save it at a resolution of 500x500. A point can be passed to personalize these dimensions
- `background` (rgb): Allows to fill the background of the display with a specific color
- `camera_interaction` (boolean): If false, the user will not be able to modify the position and the orientation of the camera, and neither using the ROI. Default is true.
- `camera_lens` (int): Allows to define the lens of the camera
- `camera_look_pos` (point): Allows to define the direction of the camera
- `camera_pos` (any type in [point, agent]): Allows to define the position of the camera
- `camera_up_vector` (point): Allows to define the orientation of the camera
- `diffuse_light` (any type in [int, rgb]): Allows to define the value of the diffuse light either using an int (diffuse_light:(125)) or a rgb color ((diffuse_light:rgb(255,255,255)). default is (127,127,127,255)
- `diffuse_light_pos` (point): Allows to define the position of the diffuse light either using an point (diffuse_light_pos:{x,y,z}). default is {world.shape.width/2,world.shape.height/2,world.shape.width*2}
- `draw_diffuse_light` (boolean): Allows to show/hide a representation of the lights. Default is false.
- `draw_env` (boolean): Allows to enable/disable the drawing of the world shape and the ordinate axes. Default can be configured in Preferences

- focus (geometry): the geometry (or agent) on which the display will (dynamically) focus
- fullscreen (boolean): Indicates whether or not the display should cover the whole screen (default is false
- light (boolean): Allows to enable/disable the light. Default is true
- orthographic_projection (boolean): Allows to enable/disable the orthographic projection. Default can be configured in Preferences
- output3D (any type in [boolean, point]):
- polygonmode (boolean):
- refresh (boolean): Indicates the condition under which this output should be refreshed (default is true)
- refresh_every (int): Allows to refresh the display every n time steps (default is 1)
- scale (any type in [boolean, float]): Allows to display a scale bar in the overlay. Accepts true/false or an unit name
- show_fps (boolean): Allows to enable/disable the drawing of the number of frames per second
- tesselation (boolean):
- trace (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied. This facet can also be applied to individual layers
- type (a label): Allows to use either Java2D (for planar models) or OpenGL (for 3D models) as the rendering subsystem
- use_shader (boolean): Under construction...
- z_fighting (boolean): Allows to alleviate a problem where agents at the same z would overlap each other in random ways

Definition

A display refers to a independent and mobile part of the interface that can display species, images, texts or charts.

Usages

- The general syntax is:

```
display my_display [additional options] { ... }
```

- Each display can include different layers (like in a GIS).

```
display gridWithElevationTriangulated type: opengl ambient_light:
    100 {
    grid cell elevation: true triangulation: true;
    species people aspect: base;
}
```

Embedments

- The `display` statement is of type: **Output**
- The `display` statement can be embedded into: output, permanent,
- The `display` statement embeds statements: agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,

---

display_grid

Facets

- `species` (species), (omissible) : the species of the agents in the grid
- `dem` (matrix):
- `draw_as_dem` (boolean):
- `elevation` (any type in [matrix, float, int, boolean]): Allows to specify the elevation of each cell, if any. Can be a matrix of float (provided it has the same size than the grid), an int or float variable of the grid species, or simply true (in which case, the variable called 'grid_value' is used to compute the elevation of each cell)
- `grayscale` (boolean): if true, givse a grey value to each polygon depending on its elevation (false by default)
- `lines` (rgb): the color to draw lines (borders of cells)
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greater than 1. The position can also be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.

- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `text` (boolean): specify whether the attribute used to compute the elevation is displayed on each cells (false by default)
- `texture` (any type in [boolean, file]): Either file containing the texture image to be applied on the grid or, if true, the use of the image composed by the colors of the cells. If false, no texture is applied
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)
- `triangulation` (boolean): specifies whther the cells will be triangulated: if it is false, they will be displayed as horizontal squares at a given elevation, whereas if it is true, cells will be triangulated and linked to neighbors in order to have a continuous surface (false by default)

Definition

`display_grid` is used using the `grid` keyword. It allows the modeler to display in an optimized way all cell agents of a grid (i.e. all agents of a species having a grid topology).

Usages

- The general syntax is:

```
display my_display {
   grid ant_grid lines: #black position: { 0.5, 0 } size:
   {0.5,0.5};
}
```

- To display a grid as a DEM:

```
display my_display {
    grid cell texture: texture_file text: false triangulation:
   true elevation: true;
}
```

- See also: display, agents, chart, event, graphics, image, overlay, display_population,

Embedments

- The `display_grid` statement is of type: **Layer**
- The `display_grid` statement can be embedded into: display,
- The `display_grid` statement embeds statements:

---

display_population

Facets

- `species` (species), (omissible) : the species to be displayed
- `aspect` (an identifier): the name of the aspect that should be used to display the species
- `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greter than 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as

absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Definition

The `display_population` statement is used using the `species keyword`. It allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

Usages

- The general syntax is:

```
display my_display {
   species species_name [additional options];
}
```

- Species can be superposed on the same plan (be careful with the order, the last one will be above all the others):

```
display my_display {
   species agent1 aspect: base;
   species agent2 aspect: base;
   species agent3 aspect: base;
}
```

- Each species layer can be placed at a different z value using the opengl display. position:{0,0,0} means the layer will be placed on the ground and position:{0,0,1} means it will be placed at an height equal to the maximum size of the environment.

```
display my_display type: opengl{
   species agent1 aspect: base ;
   species agent2 aspect: base position:{0,0,0.5};
   species agent3 aspect: base position:{0,0,1};
}
```

- See also: display, agents, chart, event, graphics, display_grid, image, overlay,

Embedments

- The `display_population` statement is of type: **Layer**
- The `display_population` statement can be embedded into: display, display_population,
- The `display_population` statement embeds statements: display_population,

---

do

Facets

- **`action`** (an identifier), (omissible) : the name of an action or a primitive
- `internal_function` (any type):
- `returns` (a new identifier): create a new variable and assign to it the result of the action
- `with` (map): a map expression containing the parameters of the action

Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this [BuiltIn161 page]; for the list of primitives defined by the different skills, see this [Skills161 page]. Finally, see this [Species161 page] to know how to declare custom actions.

Usages

- The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

```
do name_of_action_or_primitive;
```

- In case the action expects one or more arguments to be passed, they are defined by using facets (enclosed tags or a map are now deprecated):

```
do name_of_action_or_primitive arg1: expression1 arg2:
   expression2;
```

- In case the result of the action needs to be made available to the agent, the action can be called with the agent calling the action (`self` when the agent itself calls the action) instead of `do`; the result should be assigned to a temporary variable:

```
type_returned_by_action result <- self
   name_of_action_or_primitive [];
```

- In case of an action expecting arguments and returning a value, the following syntax is used:

```
type_returned_by_action result <- self
   name_of_action_or_primitive [arg1::expression1, arg2::
   expression2];
```

- Deprecated uses: following uses of the `do` statement (still accepted) are now deprecated:

```
// Simple syntax:
do action: name_of_action_or_primitive;

// In case the result of the action needs to be made available to
    the agent, the `returns` keyword can be defined; the result
   will then be referred to by the temporary variable declared in
    this attribute:
do name_of_action_or_primitive returns: result;
do name_of_action_or_primitive arg1: expression1 arg2:
   expression2 returns: result;
type_returned_by_action result <- name_of_action_or_primitive(
   self, [arg1::expression1, arg2::expression2]);

// In case the result of the action needs to be made available to
    the agent
let result <- name_of_action_or_primitive(self, []);

// In case the action expects one or more arguments to be passed,
    they can also be defined by using enclosed `arg` statements,
   or the `with` facet with a map of parameters:
do name_of_action_or_primitive with: [arg1::expression1, arg2::
   expression2];

or

do name_of_action_or_primitive {
     arg arg1 value: expression1;
     arg arg2 value: expression2;
     ...
}
```

Embedments

- The do statement is of type: **Single statement**
- The do statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The do statement embeds statements:

---

draw

Facets

- `geometry` (any type), (omissible) : any type of data (it can be geometry, image, text)
- `at` (point): location where the shape/text/icon is drawn
- `begin_arrow` (any type in [int, float]): the size of the arrow, located at the beginning of the drawn geometry
- `bitmap` (boolean): Whether to render the text in 3D or not
- `border` (any type in [rgb, boolean]): if used with a color, represents the color of the geometry border. If set to false, expresses that no border should be drawn. If not set, the borders will be drawn using the color of the geometry.
- `color` (rgb): the color to use to display the object. In case of images, will try to colorize it
- `depth` (float): (only if the display type is opengl) Add an artificial depth to the geometry previously defined (a line becomes a plan, a circle becomes a cylinder, a square becomes a cube, a polygon becomes a polyhedron with height equal to the depth value). Note: This only works if the geometry is not a point
- `empty` (boolean): a condition specifying whether the geometry is empty or full
- `end_arrow` (any type in [int, float]): the size of the arrow, located at the end of the drawn geometry
- `font` (any type in [19, string]): the font used to draw the text, if any. Applying this facet to geometries or images has no effect. You can construct here your font with the operator "font". ex : font:font("Helvetica", 20 , #plain)
- `material` (25): Set a particular material to the object (only if you are in the "use_-shader" mode).
- `perspective` (boolean): Whether to render the text in perspective or facing the user. Default is true.
- `rotate` (any type in [float, int, pair]): orientation of the shape/text/icon; can be either an int/float (angle) or a pair float::point (angle::rotation axis). The rotation axis, when expressed as an angle, is by defaut {0,0,1}
- `rounded` (boolean): specify whether the geometry have to be rounded (e.g. for squares)
- `size` (any type in [float, point]): size of the object to draw, expressed as a bounding box (width, height, depth). If expressed as a float, represents the size in the three directions.
- `texture` (any type in [string, list]): the texture(s) that should be applied to the geometry. Either a path to a file or a list of paths

Definition

draw is used in an aspect block to expresse how agents of the species will be drawn. It is evaluated each time the agent has to be drawn. It can also be used in the graphics block.

Usages

- Any kind of geometry as any location can be drawn when displaying an agent (independently of his shape)

```
aspect geometryAspect {
    draw circle(1.0) empty: !hasFood color: #orange ;
}
```

- Image or text can also be drawn

```
aspect arrowAspect {
    draw "Current state= "+state at: location + {-3,1.5} color: #
   white font: font('Default', 12, #bold) ;
    draw file(ant_shape_full) rotate: heading at: location size:
   5
}
```

- Arrows can be drawn with any kind of geometry, using begin_arrow and end_arrow facets, combined with the empty: facet to specify whether it is plain or empty

```
aspect arrowAspect {
    draw line([{20, 20}, {40, 40}]) color: #black begin_arrow:5;
    draw line([{10, 10},{20, 50}, {40, 70}]) color: #green
   end_arrow: 2 begin_arrow: 2 empty: true;
    draw square(10) at: {80,20} color: #purple begin_arrow: 2
   empty: true;
}
```

Embedments

- The `draw` statement is of type: **Single statement**
- The `draw` statement can be embedded into: aspect, Sequence of statements or action, Layer,
- The `draw` statement embeds statements:

---

else

Facets

Definition

This statement cannot be used alone

Usages

- See also: if,

Embedments

- The `else` statement is of type: **Sequence of statements or action**
- The `else` statement can be embedded into: if,
- The `else` statement embeds statements:

---

emotional_contagion

Facets

- `emotion_detected` (546706): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the emotional contagion

- charisma (float): The charisma value of the perceived agent (between 0 and 1)
- emotion_created (546706): the emotion that will be created with the contagion
- receptivity (float): The receptivity value of the current agent (between 0 and 1)
- threshold (float): The threshold value to make the contagion
- when (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to make conscious or unconscious emotional contagion

Usages

- Other examples of use:

```
emotional_contagion emotion_detected:fearConfirmed;
emotional_contagion emotion_detected:fear emotion_created:
   fearConfirmed;
emotional_contagion emotion_detected:fear emotion_created:
   fearConfirmed charisma: 0.5 receptivity: 0.5;
```

Embedments

- The emotional_contagion statement is of type: **Single statement**
- The emotional_contagion statement can be embedded into: Behavior, Sequence of statements or action,
- The emotional_contagion statement embeds statements:

---

enter

Facets

Definition

In an FSM architecture, enter introduces a sequence of statements to execute upon entering a state.

Usages

- In the following example, at the step it enters into the state s_init, the message 'Enter in s_init' is displayed followed by the display of the state name:

```
state s_init {
    enter { write "Enter in" + state; }
        write "Enter in" + state;
    }
    write state;
}
```

- See also: state, exit, transition,

Embedments

- The `enter` statement is of type: **Sequence of statements or action**
- The `enter` statement can be embedded into: state,
- The `enter` statement embeds statements:

---

equation

Facets

- `name` (an identifier), (omissible) : the equation identifier
- `params` (list): the list of parameters used in predefined equation systems
- `simultaneously` (list): a list of species containing a system of equations (all systems will be solved simultaneously)
- `type` (an identifier), takes values in: {SI, SIS, SIR, SIRS, SEIR, LV}: the choice of one among classical models (SI, SIS, SIR, SIRS, SEIR, LV)
- `vars` (list): the list of variables used in predefined equation systems

Definition

The equation statement is used to create an equation system from several single equations.

Usages

- The basic syntax to define an equation system is:

```
float t;
float S;
float I;
equation SI {
   diff(S,t) = (- 0.3 * S * I / 100);
   diff(I,t) = (0.3 * S * I / 100);
}
```

- If the type: facet is used, a predefined equation system is defined using variables vars: and parameters params: in the right order. All possible predefined equation systems are the following ones (see [EquationPresentation161 EquationPresentation161] for precise definition of each classical equation system):

```
equation eqSI type: SI vars: [S,I,t] params: [N,beta];
equation eqSIS type: SIS vars: [S,I,t] params: [N,beta,gamma];
equation eqSIR type:SIR vars:[S,I,R,t] params:[N,beta,gamma];
equation eqSIRS type: SIRS vars: [S,I,R,t] params: [N,beta,gamma,
   omega,mu];
equation eqSEIR type: SEIR vars: [S,E,I,R,t] params: [N,beta,
   gamma,sigma,mu];
equation eqLV type: LV vars: [x,y,t] params: [alpha,beta,delta,
   gamma] ;
```

- If the simultaneously: facet is used, system of all the agents will be solved simultaneously.

- See also: =, solve,

Embedments

- The `equation` statement is of type: **Sequence of statements or action**
- The `equation` statement can be embedded into: Species, Model,
- The `equation` statement embeds statements: =,

error

Facets

- `message` (string), (omissible) : the message to display in the error.

Definition

The statement makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

Usages

- Throwing an error

```
error 'This is an error raised by ' + self;
```

Embedments

- The `error` statement is of type: **Single statement**
- The `error` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `error` statement embeds statements:

---

event

Facets

- `name` (an identifier), (omissible) : the type of event captured: can be “mouse_up”, “mouse_down”, “mouse_move”, “mouse_exit”, “mouse_enter” or a character

- `action` (any type): Either a block of statements to execute in the context of the simulation or the identifier of the action to be executed. This action needs to be defined in 'global' and will receive two possible arguments: the location of the mouse in the environment and the agents under the mouse. For instance:`action myAction (point location, list selected_agents)`
- `unused` (an identifier), takes values in: {mouse_up, mouse_down, mouse_move, mouse_enter, mouse_exit}: an unused facet that serves only for the purpose of declaring the string values

Definition

`event` allows to interact with the simulation by capturing mouse or key events and doing an action. This action needs to be defined in 'global' and will receive two possible arguments: the location of the mouse in the environment and the agents under the mouse. The names of these arguments need not to be fixed: instead, the first argument of type 'point' will receive the location of the mouse, while the first argument whose type is compatible with 'container' will receive the list of agents selected.

Usages

- The general syntax is:

```
event [event_type] action: myAction;
```

- For instance:

```
global {
   // ...
   action myAction (point location, list<agent> selected_agents)
   {
      // location: contains le location of the click in the
   environment
      // selected_agents: contains agents clicked by the event

      // code written by modelers
   }
}

experiment Simple type:gui {
   display my_display {
      event mouse_up action: myAction;
   }
}
```

- See also: display, agents, chart, graphics, display_grid, image, overlay, display_population,

Embedments

- The `event` statement is of type: **Layer**
- The `event` statement can be embedded into: display,
- The `event` statement embeds statements:

---

exhaustive

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize

Definition

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way. See [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the exhaustive statement uses `method exhaustive` instead of the expected `exhaustive name: id`:

```
method exhaustive [facet: value];
```

- For example:

```
method exhaustive maximize: food_gathered;
```

Embedments

- The `exhaustive` statement is of type: **Batch method**
- The `exhaustive` statement can be embedded into: Experiment,
- The `exhaustive` statement embeds statements:

---

exit

Facets

Definition

In an FSM architecture, `exit` introduces a sequence of statements to execute right before exiting the state.

Usages

- In the following example, at the state it leaves the state s_init, he will display the message 'EXIT from s_init':

```
    state s_init initial: true {
        write state;
        transition to: s1 when: (cycle > 2) {
            write "transition s_init -> s1";
        }
        exit {
            write "EXIT from "+state;
        }
    }
```

- See also: enter, state, transition,

Embedments

- The `exit` statement is of type: **Sequence of statements or action**
- The `exit` statement can be embedded into: state,
- The `exit` statement embeds statements:

---

experiment

Facets

- `name` (a label), (omissible) : identifier of the experiment

- `title` (a label):

- `type` (a label), takes values in: {batch, memorize, gui, headless}: the type of the experiment (either 'gui' or 'batch'
- `control` (an identifier):
- `frequency` (int): the execution frequence of the experiment (default value: 1). If frequency: 10, the experiment is executed only each 10 steps.
- `keep_seed` (boolean):
- `keep_simulations` (boolean): In the case of a batch experiment, specifies whether or not the simulations should be kept in memory for further analysis or immediately discarded with only their fitness kept in memory

- `multicore` (boolean): Allows the experiment, when set to true, to use multiple threads to run its simulations
- `parent` (an identifier): the parent experiment (in case of inheritance between experiments)
- `repeat` (int): In the case of a batch experiment, expresses hom many times the simulations must be repeated
- `schedules` (container): an ordered list of agents giving the order of their execution
- `skills` (list):
- `until` (boolean): In the case of a batch experiment, an expression that will be evaluated to know when a simulation should be terminated

Embedments

- The `experiment` statement is of type: **Experiment**
- The `experiment` statement can be embedded into: Model,
- The `experiment` statement embeds statements:

---

export

Facets

- `var` (an identifier), (omissible) :
- `framerate` (int):
- `name` (string):

Embedments

- The `export` statement is of type: **Single statement**
- The `export` statement can be embedded into: Experiment,
- The `export` statement embeds statements:

---

focus

Facets

- name (an identifier), (omissible) : the identifier of the focus
- expression (any type): an expression that will be the value kept in the belief
- priority (any type in [float, int]): The priority of the created predicate
- var (any type in [any type, list, container]): the variable of the perceived agent you want to add to your beliefs
- when (boolean): A boolean value to focus only with a certain condition

Definition

enables to directly add a belief from the variable of a perceived specie.

Usages

- Other examples of use:

```
focus var:speed /*where speed is a variable from a species
   that is being perceived*/
```

Embedments

- The focus statement is of type: **Single statement**
- The focus statement can be embedded into: Behavior, Sequence of statements or action,
- The focus statement embeds statements:

---

focus_on

Facets

- value (any type), (omissible) : The agent, list of agents, geometry to focus on

Definition

Allows to focus on the passed parameter in all available displays. Passing 'nil' for the parameter will make all screens return to their normal zoom

Usages

- Focuses on an agent, a geometry, a set of agents, etc...)

```
focus_on my_species(0);
```

Embedments

- The `focus_on` statement is of type: **Single statement**
- The `focus_on` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `focus_on` statement embeds statements:

---

genetic

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `crossover_prob` (float): crossover probability between two individual solutions
- `max_gen` (int): number of generations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `mutation_prob` (float): mutation probability for an individual solution
- `nb_prelim_gen` (int): number of random populations used to build the initial population
- `pop_dim` (int): size of the population (number of individual solutions)

Definition

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article and [batch161 the batch dedicated page]. The principle of the GA is to search an optimal solution by applying evolution operators on an initial population of solutions. There are three types of evolution operators: crossover, mutation and selection. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Usages

- As other batch methods, the basic syntax of the `genetic` statement uses `method genetic` instead of the expected `genetic name: id`:

```
method genetic [facet: value];
```

- For example:

```
method genetic maximize: food_gathered pop_dim: 5 crossover_prob:
    0.7 mutation_prob: 0.1 nb_prelim_gen: 1 max_gen: 20;
```

Embedments

- The `genetic` statement is of type: **Batch method**
- The `genetic` statement can be embedded into: Experiment,
- The `genetic` statement embeds statements:

---

graphics

Facets

- `name` (a label), (omissible) : the human readable title of the graphics
- `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false

- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greter than 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Definition

`graphics` allows the modeler to freely draw shapes/geometries/texts without having to define a species. It works exactly like a species [Aspect161 aspect]: the draw statement can be used in the same way.

Usages

- The general syntax is:

```
display my_display {
   graphics "my new layer" {
      draw circle(5) at: {10,10} color: #red;
      draw "test" at: {10,10} size: 20 color: #black;
   }
}
```

- See also: display, agents, chart, event, graphics, display_grid, image, overlay, display_population,

Embedments

- The graphics statement is of type: **Layer**
- The graphics statement can be embedded into: display,
- The graphics statement embeds statements:

---

highlight

Facets

- value (agent), (omissible) : The agent to hightlight
- color (rgb): An optional color to highlight the agent. Note that this color will become the default color for further higlight operations

Definition

Allows to highlight the agent passed in parameter in all available displays, optionaly setting a color. Passing 'nil' for the agent will remove the current highlight

Usages

- Highlighting an agent

```
highlight my_species(0) color: #blue;
```

Embedments

- The highlight statement is of type: **Single statement**
- The highlight statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The highlight statement embeds statements:

---

hill_climbing

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize

Definition

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the `hill_climbing` statement uses `method hill_climbing` instead of the expected `hill_climbing name: id` :

```
method hill_climbing [facet: value];
```

- For example:

```
method hill_climbing iter_max: 50 maximize : food_gathered;
```

Embedments

- The `hill_climbing` statement is of type: **Batch method**
- The `hill_climbing` statement can be embedded into: Experiment,
- The `hill_climbing` statement embeds statements:

if

Facets

- `condition` (boolean), (omissible) : A boolean expression: the condition that is evaluated.

Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true.

Usages

- The generic syntax is:

```
if bool_expr {
    [statements]
}
```

- Optionally, the statements to execute when the condition evaluates to false can be defined in a following statement else. The syntax then becomes:

```
if bool_expr {
    [statements]
}
else {
    [statements]
}
string valTrue <- "";
if true {
    valTrue <- "true";
}
else {
    valTrue <- "false";
}
    // valTrue equals "true"
```

```
string valFalse <- "";
if false {
    valFalse <- "true";
}
else {
    valFalse <- "false";
}
    // valFalse equals "false"
```

- ifs and elses can be imbricated as needed. For instance:

```
if bool_expr {
    [statements]
}
else if bool_expr2 {
    [statements]
}
else {
    [statements]
}
```

Embedments

- The `if` statement is of type: **Sequence of statements or action**
- The `if` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `if` statement embeds statements: else,

---

image

Facets

- `name` (string), (omissible) : Human readable title of the image layer

- `color` (rgb): in the case of a shapefile, this the color used to fill in geometries of the shapefile
- `file` (any type in [string, file]): the name/path of the image (in the case of a raster image)
- `gis` (any type in [file, string]): the name/path of the shape file (to display a shapefile as background, without creating agents from it)
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greter than 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `refresh` (boolean): (openGL only) specify whether the image display is refreshed. (true by default, usefull in case of images that is not modified over the simulation)
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `transparency` (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Definition

`image` allows modeler to display an image (e.g. as background of a simulation).

Usages

- The general syntax is:

```
display my_display {
   image layer_name file: image_file [additional options];
}
```

- For instance, in the case of a bitmap image

```
display my_display {
   image background file:"../images/my_backgound.jpg";
}
```

- Or in the case of a shapefile:

```
display my_display {
   image testGIS gis: "../includes/building.shp" color: rgb('blue
   ');
}
```

- It is also possible to superpose images on different layers in the same way as for species using opengl display:

```
display my_display {
  image image1 file:"../images/image1.jpg";
  image image2 file:"../images/image2.jpg";
  image image3 file:"../images/image3.jpg" position: {0,0,0.5};
}
```

- See also: display, agents, chart, event, graphics, display_grid, overlay, display_population,

Embedments

- The `image` statement is of type: **Layer**
- The `image` statement can be embedded into: display,
- The `image` statement embeds statements:

inspect

Facets

- `name` (any type), (omissible) : the identifier of the inspector
- `attributes` (list): the list of attributes to inspect
- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)
- `refresh_every` (int): Allows to refresh the inspector every n time steps (default is 1)
- `type` (an identifier), takes values in: {agent, table}: the way to inspect agents: in a table, or a set of inspectors
- `value` (any type): the set of agents to inspect, could be a species, a list of agents or an agent

Definition

`inspect` (and `browse`) statements allows modeler to inspect a set of agents, in a table with agents and all their attributes or an agent inspector per agent, depending on the type: chosen. Modeler can choose which attributes to display. When `browse` is used, type: default value is table, whereas when`inspect` is used, type: default value is agent.

Usages

- An example of syntax is:

```
inspect "my_inspector" value: ant attributes: ["name", "location"
  ];
```

Embedments

- The `inspect` statement is of type: **Output**
- The `inspect` statement can be embedded into: output, permanent, Behavior, Sequence of statements or action,
- The `inspect` statement embeds statements:

let

Facets

- `name` (a new identifier), (omissible) :
- `index` (a datatype identifier):
- `of` (a datatype identifier):
- `type` (a datatype identifier):
- `value` (any type):

Embedments

- The `let` statement is of type: **Single statement**
- The `let` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `let` statement embeds statements:

---

light

Facets

- `id` (int), (omissible) : a number from 1 to 7 to specify which light we are using
- `active` (boolean): a boolean expression telling if you want this light to be switch on or not. (default value : true)
- `color` (any type in [int, rgb]): an int / rgb / rgba value to specify the color and the intensity of the light. (default value : (127,127,127,255)).
- `direction` (point): the direction of the light (only for direction and spot light). (default value : {0.5,0.5,-1})
- `draw_light` (boolean): draw or not the light. (default value : false).
- `linear_attenuation` (float): the linear attenuation of the positionnal light. (default value : 0)
- `position` (point): the position of the light (only for point and spot light). (default value : {0,0,1})
- `quadratic_attenuation` (float): the linear attenuation of the positionnal light. (default value : 0)

- spot_angle (float): the angle of the spot light in degree (only for spot light). (default value : 45)
- type (a label): the type of light to create. A value among {point, direction, spot}. (default value : direction)
- update (boolean): specify if the light has to be updated. (default value : true).

Definition

light allows to define diffusion lights in your 3D display.

Usages

- The general syntax is:

```
light 1 type:point position:{20,20,20} color:255,
   linear_attenuation:0.01 quadratic_attenuation:0.0001
   draw_light:true update:false
light 2 type:spot position:{20,20,20} direction:{0,0,-1} color
   :255 spot_angle:25 linear_attenuation:0.01
   quadratic_attenuation:0.0001 draw_light:true update:false
light 3 type:point direction:{1,1,-1} color:255 draw_light:true
   update:false
```

- See also: display,

Embedments

- The light statement is of type: **Layer**
- The light statement can be embedded into: display,
- The light statement embeds statements:

loop

Facets

- `name` (a new identifier), (omissible) : a temporary variable name
- `from` (int): an int expression
- `over` (any type in [container, point]): a list, point, matrix or map expression
- `step` (int): an int expression
- `times` (int): an int expression
- `to` (int): an int expression
- `while` (boolean): a boolean expression

Definition

Allows the agent to perform the same set of statements either a fixed number of times, or while a condition is true, or by progressing in a collection of elements or along an interval of integers. Be aware that there are no prevention of infinite loops. As a consequence, open loops should be used with caution, as one agent may block the execution of the whole model.

Usages

- The basic syntax for repeating a fixed number of times a set of statements is:

```
loop times: an_int_expression {
     // [statements]
}
```

- The basic syntax for repeating a set of statements while a condition holds is:

```
loop while: a_bool_expression {
     // [statements]
}
```

- The basic syntax for repeating a set of statements by progressing over a container of a point is:

```
loop a_temp_var over: a_collection_expression {
     // [statements]
}
```

- The basic syntax for repeating a set of statements while an index iterates over a range of values with a fixed step of 1 is:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 {
     // [statements]
}
```

- The incrementation step of the index can also be chosen:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 step:
    int_expression3 {
     // [statements]
}
```

- In these latter three cases, the name facet designates the name of a temporary variable, whose scope is the loop, and that takes, in turn, the value of each of the element of the list (or each value in the interval). For example, in the first instance of the "loop over" syntax :

```
int a <- 0;
loop i over: [10, 20, 30] {
     a <- a + i;
} // a now equals 60
```

- The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to facets take an integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the inclusive interval (i.e. [to, from]). If the step is not defined, it is assumed to be equal to 1 or -1, depending on the direction of the range. If it is defined, its sign will be respected, so that a positive step will never allow the loop to enter a loop from i to j where i is greater than j

```
list the_list <-list (species_of (self));
loop i from: 0 to: length (the_list) - 1 {
     ask the_list at i {
        // ...
     }
} // every  agent of the list is asked to do something
```

Embedments

- The `loop` statement is of type: **Sequence of statements or action**
- The `loop` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `loop` statement embeds statements:

---

match

Facets

- `value` (any type), (omissible) :

Definition

In a switch…match structure, the value of each match block is compared to the value in the switch. If they match, the embedded statement set is executed. Three kinds of match can be used

Usages

- match block is executed if the switch value is equals to the value of the match:

```
switch 3 {
   match 1 {write "Match 1"; }
   match 3 {write "Match 2"; }
}
```

- match_between block is executed if the switch value is in the interval given in value of the match_between:

```
switch 3 {
   match_between [1,2] {write "Match OK between [1,2]"; }
   match_between [2,5] {write "Match OK between [2,5]"; }
}
```

- match_one block is executed if the switch value is equals to one of the values of the match_one:

```
switch 3 {
   match_one [0,1,2] {write "Match OK with one of [0,1,2]"; }
   match_between [2,3,4,5] {write "Match OK with one of [2,3,4,5]
   "; }
}
```

- See also: switch, default,

Embedments

- The `match` statement is of type: **Sequence of statements or action**
- The `match` statement can be embedded into: switch,
- The `match` statement embeds statements:

---

migrate

Facets

- `source` (any type in [agent, species, container, an identifier]), (omissible) : can be an agent, a list of agents, a agent's population to be migrated

- `target` (species): target species/population that source agent(s) migrate to.
- `returns` (a new identifier): the list of returned agents in a new local variable

Definition

This command permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Species of source agents and target species respect the following constraints: (i) they are "peer" species (sharing the same direct macro-species), (ii) they have sub-species vs. parent-species relationship.

Usages

- It can be used in a 3-levels model, in case where individual agents can be captured into group meso agents and groups into clouds macro agents. migrate is used to allows agents captured by groups to migrate into clouds. See the model 'Balls, Groups and Clouds.gaml' in the library.

```
migrate ball_in_group target: ball_in_cloud;
```

- See also: capture, release,

Embedments

- The `migrate` statement is of type: **Sequence of statements or action**
- The `migrate` statement can be embedded into: Behavior, Sequence of statements or action,
- The `migrate` statement embeds statements:

---

monitor

Facets

- `name` (a label), (omissible) : identifier of the monitor

- `value` (any type): expression that will be evaluated to be displayed in the monitor
- `color` (rgb): Indicates the (possibly dynamic) color of this output (default is a light gray)

- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)
- `refresh_every` (int): Allows to refresh the monitor every n time steps (default is 1)

Definition

A monitor allows to follow the value of an arbitrary expression in GAML.

Usages

- An example of use is:

```
monitor "nb preys" value: length(prey as list) refresh_every: 5;
```

Embedments

- The `monitor` statement is of type: **Output**
- The `monitor` statement can be embedded into: output, permanent,
- The `monitor` statement embeds statements:

---

output

Facets

Definition

`output` blocks define how to visualize a simulation (with one or more display blocks that define separate windows). It will include a set of displays, monitors and files statements. It will be taken into account only if the experiment type is `gui`.

Usages

- Its basic syntax is:

```
experiment exp_name type: gui {
   // [inputs]
   output {
      // [display, file, inspect, layout or monitor statements]
   }
}
```

- See also: display, monitor, inspect, output_file, layout,

Embedments

- The `output` statement is of type: **Output**
- The `output` statement can be embedded into: Model, Experiment,
- The `output` statement embeds statements: display, inspect, monitor, output_file,

---

output_file

Facets

- `name` (an identifier), (omissible) : The name of the file where you want to export the data

- `data` (string): The data you want to export
- `footer` (string): Define a footer for your export file
- `header` (string): Define a header for your export file
- `refresh` (boolean): Indicates the condition under which this file should be saved (default is true)
- `refresh_every` (int): Allows to save the file every n time steps (default is 1)
- `rewrite` (boolean): Rewrite or not the existing file
- `type` (an identifier), takes values in: {csv, text, xml}: The type of your output data

Embedments

- The `output_file` statement is of type: **Output**
- The `output_file` statement can be embedded into: output, permanent,
- The `output_file` statement embeds statements:

---

overlay

Facets

- `background` (rgb): the background color of the overlay displayed inside the view (the bottom overlay remains black)
- `border` (rgb): Color to apply to the border of the rectangular shape of the overlay. Nil by default
- `center` (any type): an expression that will be evaluated and displayed in the center section of the bottom overlay
- `color` (any type in [list, rgb]): the color(s) used to display the expressions given in the 'left', 'center' and 'right' facets
- `left` (any type): an expression that will be evaluated and displayed in the left section of the bottom overlay
- `position` (point): position of the upper-left corner of the overlay. Note that if coordinates are in [0,1[, the position is relative to the size of the view (e.g. {0.5,0.5} refers to the middle of the view) whereas it is absolute when coordinates are greater than 1. When the position is a 3D point {0.5, 0.5, 0.5}, the last coordinate specifies the elevation of the layer.
- `right` (any type): an expression that will be evaluated and displayed in the right section of the bottom overlay
- `rounded` (boolean): Whether or not the rectangular shape of the overlay should be rounded. True by default
- `size` (point): extent of the layer in the view from its position. Coordinates in [0,1[are treated as percentages of the total surface of the view, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Unlike 'position', no elevation can be provided with the z coordinate
- `transparency` (float): the transparency rate of the overlay (between 0 and 1, 1 means no transparency) when it is displayed inside the view. The bottom overlay will remain at 0.75

Definition

overlay allows the modeler to display a line to the already existing bottom overlay, where the results of 'left', 'center' and 'right' facets, when they are defined, are displayed with the corresponding color if defined.

Usages

- To display information in the bottom overlay, the syntax is:

```
overlay "Cycle: " + (cycle) center: "Duration: " + total_duration
    + "ms" right: "Model time: " + as_date(time,"") color: [#
   yellow, #orange, #yellow];
```

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

Embedments

- The overlay statement is of type: **Layer**
- The overlay statement can be embedded into: display,
- The overlay statement embeds statements:

---

parameter

Facets

- var (an identifier): the name of the variable (that should be declared in the global)
- name (a label), (omissible) : The message displayed in the interface
- among (list): the list of possible values
- category (a label): a category label, used to group parameters in the interface
- init (any type): the init value
- max (any type): the maximum value

- `min` (any type): the minimum value
- `step` (float): the increment step (mainly used in batch mode to express the variation step between simulation)
- `type` (a datatype identifier): the variable type
- `unit` (a label): the variable unit

Definition

The parameter statement specifies which global attributes (i) will change through the successive simulations (in batch experiments), (ii) can be modified by user via the interface (in gui experiments). In GUI experiments, parameters are displayed depending on their type.

Usages

- In gui experiment, the general syntax is the following:

```
parameter title var: global_var category: cat;
```

- In batch experiment, the two following syntaxes can be used to describe the possible values of a parameter:

```
parameter 'Value of toto:' var: toto among: [1, 3, 7, 15, 100];
parameter 'Value of titi:' var: titi min: 1 max: 100 step: 2;
```

Embedments

- The `parameter` statement is of type: **Parameter**
- The `parameter` statement can be embedded into: Experiment,
- The `parameter` statement embeds statements:

perceive

Facets

- target (any type in [container, point, agent]): the list of the agent you want to perceive
- name (an identifier), (omissible) : the name of the perception
- as (species): an expression that evaluates to a species
- emotion (546706): The emotion needed to do the perception
- in (any type in [float, geometry]): a float or a geometry. If it is a float, it's a radius of a detection area. If it is a geometry, it is the area of detection of others species.
- threshold (float): Threshold linked to the emotion.
- when (boolean): a boolean to tell when does the perceive is active

Definition

Allow the agent, with a bdi architecture, to perceive others agents

Usages

- the basic syntax to perceive agents inside a circle of perception

```
perceive name_of-perception target:
   the_agents_you_want_to_perceive in: a_distance when:
   a_certain_condition {
Here you are in the context of the perceived agents. To refer to
   the agent who does the perception, use myself.
If you want to make an action (such as adding a belief for
   example), use ask myself{ do the_action}
}
```

Embedments

- The perceive statement is of type: **Sequence of statements or action**
- The perceive statement can be embedded into: Species, Model,
- The perceive statement embeds statements:

permanent

Facets

- `layout` (int), (omissible) : Either #none, to indicate that no layout will be imposed, or one of the four possible predefined layouts: #stack, #split, #horizontal or #vertical. This layout will be applied to both experiment and simulation display views

Definition

Represents the outputs of the experiment itself. In a batch experiment, the permanent section allows to define an output block that will NOT be re-initialized at the beginning of each simulation but will be filled at the end of each simulation.

Usages

- For instance, this permanent section will allow to display for each simulation the end value of the food_gathered variable:

```
permanent {
    display Ants background: rgb('white') refresh_every: 1 {
        chart "Food Gathered" type: series {
            data "Food" value: food_gathered;
        }
    }
}
```

Embedments

- The `permanent` statement is of type: **Output**
- The `permanent` statement can be embedded into: Experiment,
- The `permanent` statement embeds statements: display, inspect, monitor, output_file,

plan

Facets

- `name` (an identifier), (omissible) :
- `emotion` (546706):
- `finished_when` (boolean):
- `instantaneous` (boolean):
- `intention` (546704):
- `priority` (float):
- `threshold` (float):
- `when` (boolean):

Embedments

- The `plan` statement is of type: **Behavior**
- The `plan` statement can be embedded into: Species, Model,
- The `plan` statement embeds statements:

---

put

Facets

- `in` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression
- `all` (any type): any expression
- `at` (any type): any expression
- `edge` (any type): Indicates that the item to put should be considered as an edge of the receiving graph. Soon to be deprecated, use 'put edge(item)...' instead
- `key` (any type): any expression
- `weight` (float): an expression that evaluates to a float

Definition

Allows the agent to replace a value in a container at a given position (in a list or a map) or for a given key (in a map). Note that the behavior and the type of the attributes depends on the specific kind of container.

Usages

- The allowed parameters configurations are the following ones:

```
put expr at: expr in: expr_container;
put all: expr in: expr_container;
```

- In the case of a list, the position should an integer in the bound of the list. The facet all: is used to replace all the elements of the list by the given value.

```
list<int> putList <- [1,2,3,4,5];   // putList equals [1,2,3,4,5]
put -10 at: 1 in: putList;  // putList equals [1,-10,3,4,5]
put 10 all: true in: putList;   // putList equals
    [10,10,10,10,10]
```

- In the case of a matrix, the position should be a point in the bound of the matrix. The facet all: is used to replace all the elements of the matrix by the given value.

```
matrix<int> putMatrix <- matrix([[0,1],[2,3]]);     // putMatrix
    equals matrix([[0,1],[2,3]])
put -10 at: {1,1} in: putMatrix;    // putMatrix equals matrix
    ([[0,1],[2,-10]])
put 10 all: true in: putMatrix;     // putMatrix equals matrix
    ([[10,10],[10,10]])
```

- In the case of a map, the position should be one of the key values of the map. Notice that if the given key value does not exist in the map, the given pair key::value will be added to the map. The facet all is used to replace the value of all the pairs of the map.

```
map<string,int> putMap <- ["x"::4,"y"::7];  // putMap equals ["x
   "::4,"y"::7]
put -10 key: "y" in: putMap;    // putMap equals ["x"::4,"y
   "::-10]
put -20 key: "z" in: putMap;    // putMap equals ["x"::4,"y
   "::-10, "z"::-20]
put -30 all: true in: putMap;   // putMap equals ["x"::-30,"y
   "::-30, "z"::-30]
```

Embedments

- The put statement is of type: **Single statement**
- The put statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The put statement embeds statements:

---

reactive_tabu

Facets

- name (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the agregation method
- cycle_size_max (int): minimal size of the considered cycles
- cycle_size_min (int): maximal size of the considered cycles
- iter_max (int): number of iterations
- maximize (float): the value the algorithm tries to maximize
- minimize (float): the value the algorithm tries to minimize
- nb_tests_wthout_col_max (int): number of movements without collision before shortening the tabu list
- tabu_list_size_init (int): initial size of the tabu list
- tabu_list_size_max (int): maximal size of the tabu list
- tabu_list_size_min (int): minimal size of the tabu list

Definition

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle. See [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the reactive_tabu statement uses `method reactive_tabu` instead of the expected `reactive_tabu name: id` :

```
method reactive_tabu [facet: value];
```

- For example:

```
method reactive_tabu iter_max: 50 tabu_list_size_init: 5
   tabu_list_size_min: 2 tabu_list_size_max: 10
   nb_tests_wthout_col_max: 20 cycle_size_min: 2 cycle_size_max:
   20 maximize: food_gathered;
```

Embedments

- The `reactive_tabu` statement is of type: **Batch method**
- The `reactive_tabu` statement can be embedded into: Experiment,
- The `reactive_tabu` statement embeds statements:

reflex

Facets

- `name` (an identifier), (omissible) : the identifier of the reflex
- `when` (boolean): an expression that evaluates a boolean, the condition to fulfill in order to execute the statements embedded in the reflex.

Definition

A reflex is a sequence of statements that can be executed, at each time step, by the agent. If no facet when: is defined, it will be executed every time step. If there is a when: facet, it is executed only if the boolean expression evaluates to true.

Usages

- Example:

```
reflex my_reflex when: flip (0.5){      //Only executed when flip
    returns true
    write "Executing the unconditional reflex";
}
```

Embedments

- The `reflex` statement is of type: **Behavior**
- The `reflex` statement can be embedded into: Species, Experiment, Model,
- The `reflex` statement embeds statements:

---

release

Facets

- `target` (any type in [agent, list]), (omissible) : an expression that is evaluated as an agent or a list of the agents to be released

- as (species): an expression that is evaluated as a species in which the micro-agent will be released
- in (agent): an expression that is evaluated as an agent that will be the macro-agent in which micro-agent will be released, i.e. their new host
- returns (a new identifier): a new variable containing a list of the newly released agent(s)

Definition

Allows an agent to release its micro-agent(s). The preliminary for an agent to release its micro-agents is that species of these micro-agents are sub-species of other species (cf. [Species161#Nesting_species Nesting species]). The released agents won't be micro-agents of the calling agent anymore. Being released from a macro-agent, the micro-agents will change their species and host (macro-agent).

Usages

- We consider the following species. Agents of "C" species can be released from a "B" agent to become agents of "A" species. Agents of "D" species cannot be released from the "A" agent because species "D" has no parent species.

```
species A {
...
}
species B {
...
   species C parent: A {
   ...
   }
   species D {
   ...
   }
...
}
```

- To release all "C" agents from a "B" agent, agent "C" has to execute the following statement. The "C" agent will change to "A" agent. The won't consider "B" agent as their macro-agent (host) anymore. Their host (macro-agent) will the be the host (macro-agent) of the "B" agent.

```
release list(C);
```

- The modeler can specify the new host and the new species of the released agents:

```
release list (C) as: new_species in: new host;
```

- See also: capture,

Embedments

- The `release` statement is of type: **Sequence of statements or action**
- The `release` statement can be embedded into: Behavior, Sequence of statements or action,
- The `release` statement embeds statements:

---

remove

Facets

- `from` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression to remove from the container
- `all` (any type): an expression that evaluates to a container. If it is true and if the value a list, it removes the first instance of each element of the list. If it is true and the value is not a container, it will remove all instances of this value.
- `edge` (any type): Indicates that the item to remove should be considered as an edge of the receiving graph
- `index` (any type): any expression, the key at which to remove the element from the container

- key (any type): any expression, the key at which to remove the element from the container
- node (any type): Indicates that the item to remove should be considered as a node of the receiving graph
- vertex (any type):

Definition

Allows the agent to remove an element from a container (a list, matrix, map…).

Usages

- This statement should be used in the following ways, depending on the kind of container used and the expected action on it:

```
remove expr from: expr_container;
remove index: expr from: expr_container;
remove key: expr from: expr_container;
remove all: expr from: expr_container;
```

- In the case of list, the facet item: is used to remove the first occurence of a given expression, whereas all is used to remove all the occurrences of the given expression.

```
list<int> removeList <- [3,2,1,2,3];
remove 2 from: removeList;  // removeList equals [3,1,2,3]
remove 3 all: true from: removeList;    // removeList equals
    [1,2]
remove index: 1 from: removeList;   // removeList equals [1]
```

- In the case of map, the facet key: is used to remove the pair identified by the given key.

```
map<string,int> removeMap <- ["x"::5, "y"::7, "z"::7];
remove key: "x" from: removeMap;    // removeMap equals ["y"::7,
    "z"::7]
remove 7 all: true from: removeMap;     // removeMap equals map
    ([])
```

- In addition, a map a be managed as a list with pair key as index. Given that, facets item:, all: and index: can be used in the same way:

```
map<string,int> removeMapList <- ["x"::5, "y"::7, "z"::7, "t"
   ::5];
remove 7 from: removeMapList;   // removeMapList equals ["x"::5,
   "z"::7, "t"::5]
remove [5,7] all: true from: removeMapList;     // removeMapList
   equals ["t"::5]
remove index: "t" from: removeMapList;  // removeMapList equals
   map([])
```

- In the case of a graph, both edges and nodes can be removes using node: and edge facets. If a node is removed, all edges to and from this node are also removed.

```
graph removeGraph <- as_edge_graph([{1,2}::{3,4},{3,4}::{5,6}]);
remove node: {1,2} from: removeGraph;
remove node(1,2) from: removeGraph;
list var <- removeGraph.vertices;   // var equals [{3,4},{5,6}]
list var <- removeGraph.edges;  // var equals [polyline
   ({3,4}::{5,6})]
remove edge: {3,4}::{5,6} from: removeGraph;
remove edge({3,4},{5,6}) from: removeGraph;
list var <- removeGraph.vertices;   // var equals [{3,4},{5,6}]
list var <- removeGraph.edges;  // var equals []
```

- In the case of an agent or a shape, `remove` allows to remove an attribute from the attributes map of the receiver. However, for agents, it will only remove attributes that have been added dynamically, not the ones defined in the species or in its built-in parent.

```
global {
   init {
      create speciesRemove;
      speciesRemove sR <- speciesRemove(0);     // sR.a now
   equals 100
      remove key:"a" from: sR;  // sR.a now equals nil
```

```
    }
}

species speciesRemove {
    int a <- 100;
}
```

- This statement can not be used on *matrix*.

- See also: add, put,

Embedments

- The `remove` statement is of type: **Single statement**
- The `remove` statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The `remove` statement embeds statements:

---

return

Facets

- `value` (any type), (omissible) : an expression that is returned

Definition

Allows to immediately stop and tell which value to return from the evaluation of the surrounding action or top-level statement (reflex, init, etc.). Usually used within the declaration of an action. For more details about actions, see the following [Section161 section].

Usages

- Example:

```
string foo {
     return "foo";
}

reflex {
    string foo_result <- foo();      // foos_result is now equals
   to "foo"
}
```

- In the specific case one wants an agent to ask another agent to execute a statement with a return, it can be done similarly to:

```
// In Species A:
string foo_different {
     return "foo_not_same";
}
/// ....
// In Species B:
reflex writing {
    string temp <- some_agent_A.foo_different [];   // temp is
   now equals to "foo_not_same"
}
```

Embedments

- The `return` statement is of type: **Single statement**
- The `return` statement can be embedded into: action, Behavior, Sequence of statements or action,
- The `return` statement embeds statements:

rule

Facets

- `name` (an identifier), (omissible) : The name of the rule
- `belief` (546704): The mandatory belief
- `desire` (546704): The mandatory desire
- `emotion` (546706): The mandatory emotion
- `new_belief` (546704): The belief that will be added
- `new_desire` (546704): The desire that will be added
- `new_emotion` (546706): The emotion that will be added
- `new_uncertainty` (546704): The uncertainty that will be added
- `priority` (any type in [float, int]): The priority of the predicate added as a desire
- `remove_belief` (546704): The belief that will be removed
- `remove_desire` (546704): The desire that will be removed
- `remove_emotion` (546706): The emotion that will be removed
- `remove_intention` (546704): The intention that will be removed
- `remove_uncertainty` (546704): The uncertainty that will be removed
- `threshold` (float): Threshold linked to the emotion.
- `uncertainty` (546704): The mandatory uncertainty
- `when` (boolean):

Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

Usages

- Other examples of use:

```
rule belief: new_predicate("test") when: flip(0.5)
   new_desire: new_predicate("test")
```

Embedments

- The `rule` statement is of type: **Single statement**

- The `rule` statement can be embedded into: Species, Model,
- The `rule` statement embeds statements:

---

run

Facets

- `name` (string), (omissible) :

- `of` (string):
- `core` (int):
- `end_cycle` (int):
- `seed` (int):
- `with_output` (map):
- `with_param` (map):

Embedments

- The `run` statement is of type: **Sequence of statements or action**
- The `run` statement can be embedded into: Behavior, Single statement, Species, Model,
- The `run` statement embeds statements:

---

save

Facets

- `to` (string): an expression that evaluates to an string, the path to the file
- `data` (any type), (omissible) : any expression, that will be saved in the file
- `crs` (any type): the name of the projection, e.g. crs:"EPSG:4326" or its EPSG id, e.g. crs:4326. Here a list of the CRS codes (and EPSG id): http://spatialreference.org
- `header` (boolean): an expression that evaluates to a boolean, specifying whether the save will write a header if the file does not exist

- `rewrite` (boolean): an expression that evaluates to a boolean, specifying whether the save will ecrase the file or append data at the end of it
- `type` (an identifier), takes values in: {shp, text, csv, asc, geotiff, image}: an expression that evaluates to an string, the type of the output file (it can be only "shp", "asc", "geotiff", "image", "text" or "csv")
- `with` (map): Not yet used

Definition

Allows to save data in a file. The type of file can be "shp", "asc", "geotiff", "text" or "csv".

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->"  + name + ":" + location) to: "
   save_data.txt" type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the values of all attributes of all the agents of a species into a csv (with optional attributes):

```
save species_of(self) to: "save_csvfile.csv" type: "csv" header:
   false;
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp" with:
   [name::"nameAgent", location::"locationAgent"] crs: "EPSG:4326
   ";
```

- To save the grid_value attributes of all the cells of a grid into an ESRI ASCII Raster file:

```
save grid to: "save_grid.asc" type: "asc";
```

- To save the grid_value attributes of all the cells of a grid into geotiff:

```
save grid to: "save_grid.tif" type: "geotiff";
```

- To save the grid_value attributes of all the cells of a grid into png (with a worldfile):

```
save grid to: "save_grid.png" type: "image";
```

- The save statement can be use in an init block, a reflex, an action or in a user command. Do not use it in experiments.

Embedments

- The `save` statement is of type: **Single statement**
- The `save` statement can be embedded into: Behavior, Sequence of statements or action,
- The `save` statement embeds statements:

save_batch

Facets

- `to` (a label):
- `data` (any type), (omissible) :
- `rewrite` (boolean):

Embedments

- The `save_batch` statement is of type: **Batch method**
- The `save_batch` statement can be embedded into: Experiment,
- The `save_batch` statement embeds statements:

---

set

Facets

- `name` (any type), (omissible) : the name of an existing variable or attribute to be modified

- `value` (any type): the value to affect to the variable or attribute

Definition

Allows to assign a value to the variable or attribute specified

Usages

Embedments

- The `set` statement is of type: **Single statement**
- The `set` statement can be embedded into: chart, Behavior, Sequence of statements or action,

- The set statement embeds statements:

---

setup

Facets

Definition

The setup statement is used to define the set of instructions that will be executed before every [#test test].

Usages

- As every test should be independent from the others, the setup will mainly contain initialization of variables that will be used in each test.

```
species Tester {
    int val_to_test;

    setup {
        val_to_test <- 0;
    }

    test t1 {
       // [set of instructions, including asserts]
    }
}
```

- See also: test, assert,

Embedments

- The setup statement is of type: **Sequence of statements or action**
- The setup statement can be embedded into: Species, Experiment, Model,

- The setup statement embeds statements:

---

simulate

Facets

- comodel (file), (omissible) :
- repeat (int):
- reset (boolean):
- share (list):
- until (boolean):
- with_experiment (string):
- with_input (map):
- with_output (map):

Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. It obeys the following syntax, where the target attribute denotes the receiver agent(s):

Usages

- Other examples of use:

```
ask receiver_agent(s) {
 // [statements]
}
```

Embedments

- The simulate statement is of type: **Single statement**
- The simulate statement can be embedded into: chart, Experiment, Species, Behavior, Sequence of statements or action,

- The `simulate` statement embeds statements:

---

solve

Facets

- **`equation`** (an identifier), (omissible) : the equation system identifier to be numerically solved
- `cycle_length` (int): length of simulation cycle which will be synchronize with step of integrator (default value: 1)
- `discretizing_step` (int): number of discrete between 2 steps of simulation (default value: 0)
- `integrated_times` (list): time interval inside integration process
- `integrated_values` (list): list of variables's value inside integration process
- `max_step` (float): maximal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- `method` (an identifier), takes values in: {Euler, ThreeEighthes, Midpoint, Gill, Luther, rk4, dp853, AdamsBashforth, AdamsMoulton, DormandPrince54, GraggBulirschStoer, HighamHall54}: integrate method (can be only "Euler", "ThreeEighthes", "Midpoint", "Gill", "Luther", "rk4" or "dp853", "AdamsBashforth", "AdamsMoulton", "DormandPrince54", "GraggBulirschStoer", "HighamHall54") (default value: "rk4")
- `min_step` (float): minimal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- `scalAbsoluteTolerance` (float): allowed absolute error (used with dp853 method only)
- `scalRelativeTolerance` (float): allowed relative error (used with dp853 method only)
- `step` (float): integration step, use with most integrator methods (default value: 1)
- `time_final` (float): target time for the integration (can be set to a value smaller than t0 for backward integration)
- `time_initial` (float): initial time

Definition

Solves all equations which matched the given name, with all systems of agents that should solved simultaneously.

Usages

- Other examples of use:

```
solve SIR method: "rk4" step:0.001;
```

Embedments

- The `solve` statement is of type: **Single statement**
- The `solve` statement can be embedded into: Behavior, Sequence of statements or action,
- The `solve` statement embeds statements:

---

species

Facets

- `name` (an identifier), (omissible) : the identifier of the species
- `cell_height` (float): (grid only), the height of the cells of the grid
- `cell_width` (float): (grid only), the width of the cells of the grid
- `compile` (boolean):
- `control` (22): defines the architecture of the species (e.g. fsm...)
- `edge_species` (species): In the case of a species defining a graph topology for its instances (nodes of the graph), specifies the species to use for representing the edges
- `file` (file): (grid only), a bitmap file that will be loaded at runtime so that the value of each pixel can be assigned to the attribute 'grid_value'
- `frequency` (int): The execution frequency of the species (default value: 1). For instance, if frequency is set to 10, the population of agents will be executed only every 10 cycles.

- `height` (int): (grid only), the height of the grid (in terms of agent number)
- `mirrors` (any type in [list, species]): The species this species is mirroring. The population of this current species will be dependent of that of the species mirrored (i.e. agents creation and death are entirely taken in charge by GAMA with respect to the demographics of the species mirrored). In addition, this species is provided with an attribute called 'target', which allows each agent to know which agent of the mirrored species it is representing.
- `neighbors` (int): (grid only), the chosen neighborhood (4, 6 or 8)
- `neighbours` (int): (grid only), the chosen neighborhood (4, 6 or 8)
- `parent` (species): the parent class (inheritance)
- `schedules` (container): A container of agents (a species, a dynamic list, or a combination of species and containers) , which represents which agents will be actually scheduled when the population is scheduled for execution. For instance, 'species a schedules: (10 among a)' will result in a population that schedules only 10 of its own agents every cycle. 'species b schedules: ' will prevent the agents of 'b' to be scheduled. Note that the scope of agents covered here can be larger than the population, which allows to build complex scheduling controls; for instance, defining 'global schedules: {…} species b schedules: ; species c schedules: b + world;' allows to simulate a model where the agents of b are scheduled first, followed by the world, without even having to create an instance of c.
- `skills` (list): The list of skills that will be made available to the instances of this species. Each new skill provides attributes and actions that will be added to the ones defined in this species
- `topology` (topology): The topology of the population of agents defined by this species. In case of nested species, it can for example be the shape of the macro-agent. In case of grid or graph species, the topology is automatically computed and cannot be redefined
- `torus` (boolean): is the topology toric (defaut: false). Needs to be defined on the global species.
- `use_individual_shapes` (boolean): (grid only),(true by default). Allows to specify whether or not the agents of the grid will have distinct geometries. If set to false, they will all have simpler proxy geometries
- `use_neighbors_cache` (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbors cache used for grids. Note that if a diffusion of variable occurs, GAMA will emit a warning and automatically switch to a caching version
- `use_regular_agents` (boolean): (grid only),(true by default). Allows to specify if the agents of the grid are regular agents (like those of any other species) or minimal ones (which can't have sub-populations, can't inherit from a regular species, etc.)
- `width` (int): (grid only), the width of the grid (in terms of agent number)

Definition

The species statement allows modelers to define new species in the model. `global` and `grid` are speciel cases of species: `global` being the definition of the global agent (which has automatically one instance, world) and `grid` being a species with a grid topology.

Usages

- Here is an example of a species definition with a FSM architecture and the additional skill moving:

```
species ant skills: [moving] control: fsm {
```

- In the case of a species aiming at mirroring another one:

```
species node_agent mirrors: list(bug) parent: graph_node
   edge_species: edge_agent {
```

- The definition of the single grid of a model will automatically create gridwidth x gridheight agents:

```
grid ant_grid width: gridwidth height: gridheight file: grid_file
    neighbors: 8 use_regular_agents: false {
```

- Using a file to initialize the grid can replace width/height facets:

```
grid ant_grid file: grid_file neighbors: 8 use_regular_agents:
   false {
```

Embedments

- The `species` statement is of type: **Species**
- The `species` statement can be embedded into: Model, Environment, Species,
- The `species` statement embeds statements:

state

Facets

- `name` (an identifier), (omissible) : the identifier of the state
- `final` (boolean): specifies whether the state is a final one (i.e. there is no transition from this state to another state) (default value= false)
- `initial` (boolean): specifies whether the state is the initial one (default value = false)

Definition

A state, like a reflex, can contains several statements that can be executed at each time step by the agent.

Usages

- Here is an exemple integrating 2 states and the statements in the FSM architecture:

```
    state s_init initial: true {
        enter { write "Enter in" + state; }
            write "Enter in" + state;
        }

        write state;

        transition to: s1 when: (cycle > 2) {
            write "transition s_init -> s1";
        }

        exit {
            write "EXIT from "+state;
        }
    }
    state s1 {

    enter {write 'Enter in '+state;}

    write state;
    exit {write 'EXIT from '+state;}
}
```

- See also: enter, exit, transition,

Embedments

- The `state` statement is of type: **Behavior**
- The `state` statement can be embedded into: fsm, Species, Experiment, Model,
- The `state` statement embeds statements: enter, exit,

---

status

Facets

- `message` (any type), (omissible) : Allows to display a necessarily short message in the status box in the upper left corner. No formatting characters (carriage returns, tabs, or Unicode characters) should be used, but a background color can be specified. The message will remain in place until it is replaced by another one or by nil, in which case the standard status (number of cycles) will be displayed again
- `color` (rgb):

Definition

The statement makes the agent output an arbitrary message in the status box.

Usages

- Outputting a message

```
status ('This is my status ' + self) color: Ã Â°yellow;
```

Embedments

- The `status` statement is of type: **Single statement**
- The `status` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `status` statement embeds statements:

---

switch

Facets

- `value` (any type), (omissible) : an expression

Definition

The "switch… match" statement is a powerful replacement for imbricated "if … else …" constructs. All the blocks that match are executed in the order they are defined. The block prefixed by default is executed only if none have matched (otherwise it is not).

Usages

- The prototypical syntax is as follows:

```
switch an_expression {
        match value1 {...}
        match_one [value1, value2, value3] {...}
        match_between [value1, value2] {...}
        default {...}
}
```

- Example:

```
switch 3 {
   match 1 {write "Match 1"; }
   match 2 {write "Match 2"; }
   match 3 {write "Match 3"; }
   match_one [4,4,6,3,7]  {write "Match one_of"; }
   match_between [2, 4] {write "Match between"; }
   default {write "Match Default"; }
}
```

- See also: match, default, if,

Embedments

- The `switch` statement is of type: **Sequence of statements or action**
- The `switch` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `switch` statement embeds statements: default, match,

---

tabu

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `tabu_list_size` (int): size of the tabu list

Definition

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the tabu statement uses `method tabu` instead of the expected `tabu name: id`:

```
method tabu [facet: value];
```

- For example:

```
method tabu iter_max: 50 tabu_list_size: 5 maximize:
    food_gathered;
```

Embedments

- The `tabu` statement is of type: **Batch method**
- The `tabu` statement can be embedded into: Experiment,
- The `tabu` statement embeds statements:

---

task

Facets

- `name` (an identifier), (omissible) : the identifier of the task

- `weight` (float): the priority level of the task

Definition

As reflex, a task is a sequence of statements that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler chooses a task to execute based on its current priority weight value.

Usages

Embedments

- The `task` statement is of type: **Behavior**
- The `task` statement can be embedded into: weighted_tasks, sorted_tasks, probabilistic_tasks, Species, Experiment, Model,
- The `task` statement embeds statements:

---

test

Facets

- `name` (an identifier), (omissible) : identifier of the test

Definition

The test statement allows modeler to define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue (if GAMA is configured in the preferences that the program does not stop at the first exception).

Usages

- An example of use:

```
species Tester {
    // set of attributes that will be used in test

    setup {
        // [set of instructions... in particular initializations]
    }

    test t1 {
        // [set of instructions, including asserts]
    }
}
```

- See also: setup, assert,

Embedments

- The `test` statement is of type: **Behavior**
- The `test` statement can be embedded into: Species, Experiment, Model,
- The `test` statement embeds statements: assert,

---

trace

Facets

Definition

All the statements executed in the trace statement are displayed in the console.

Usages

Embedments

- The `trace` statement is of type: **Sequence of statements or action**
- The `trace` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `trace` statement embeds statements:

---

transition

Facets

- to (an identifier): the identifier of the next state
- when (boolean), (omissible) : a condition to be fulfilled to have a transition to another given state

Definition

In an FSM architecture, `transition` specifies the next state of the life cycle. The transition occurs when the condition is fulfilled. The embedded statements are executed when the transition is triggered.

Usages

- In the following example, the transition is executed when after 2 steps:

```
state s_init initial: true {
    write state;
    transition to: s1 when: (cycle > 2) {
        write "transition s_init -> s1";
    }
}
```

- See also: enter, state, exit,

Embedments

- The `transition` statement is of type: **Sequence of statements or action**
- The `transition` statement can be embedded into: Sequence of statements or action, Behavior,
- The `transition` statement embeds statements:

unconscious_contagion

Facets

- emotion (546706): the emotion that will be copied with the contagion
- name (an identifier), (omissible) : the identifier of the unconscious contagion
- charisma (float): The charisma value of the perceived agent (between 0 and 1)
- receptivity (float): The receptivity value of the current agent (between 0 and 1)
- threshold (float): The threshold value to make the contagion
- when (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly copy an emotion presents in the perceived specie.

Usages

- Other examples of use:

```
unconscious_contagion emotion:fearConfirmed;
unconscious_contagion emotion:fearConfirmed charisma: 0.5
   receptivity: 0.5;
```

Embedments

- The unconscious_contagion statement is of type: **Single statement**
- The unconscious_contagion statement can be embedded into: Behavior, Sequence of statements or action,
- The unconscious_contagion statement embeds statements:

---

user_command

Facets

- name (a label), (omissible) : the identifier of the user_command

- action (an identifier): the identifier of the action to be executed. This action should be accessible in the context in which it is defined (an experiment, the global section or a species). A special case is allowed to maintain the compatibility with older versions of GAMA, when the user_command is declared in an experiment and the action is declared in 'global'. In that case, all the simulations managed by the experiment will run the action in response to the user executing the command
- color (rgb): The color of the button to display
- continue (boolean): Whether or not the button, when clicked, should dismiss the user panel it is defined in. Has no effect in other contexts (menu, parameters, inspectors)
- when (boolean): the condition that should be fulfilled (in addition to the user clicking it) in order to execute this action
- with (map): the map of the parameters::values required by the action

Definition

Anywhere in the global block, in a species or in an (GUI) experiment, user_command statements allows to either call directly an existing action (with or without arguments) or to be followed by a block that describes what to do when this command is run.

Usages

- The general syntax is for example:

```
user_command kill_myself action: some_action with: [arg1::val1,
   arg2::val2, ...];
```

- See also: user_init, user_panel, user_input,

Embedments

- The user_command statement is of type: **Sequence of statements or action**
- The user_command statement can be embedded into: user_panel, Species, Experiment, Model,
- The user_command statement embeds statements: user_input,

user_init

Facets

- `name` (an identifier), (omissible) :
- `initial` (boolean):

Definition

Used in the user control architecture, user_init is executed only once when the agent is created. It opens a special panel (if it contains user_commands statements). It is the equivalent to the init block in the basic agent architecture.

Usages

- See also: user_command, user_init, user_input,

Embedments

- The `user_init` statement is of type: **Behavior**
- The `user_init` statement can be embedded into: Species, Experiment, Model,
- The `user_init` statement embeds statements: user_panel,

---

user_input

Facets

- `returns` (a new identifier): a new local variable containing the value given by the user
- `name` (a label), (omissible) : the displayed name
- `among` (list): the set of acceptable values for the variable
- `init` (any type): the init value
- `max` (float): the maximum value
- `min` (float): the minimum value
- `type` (a datatype identifier): the variable type

Definition

It allows to let the user define the value of a variable.

Usages

- Other examples of use:

```
user_panel "Advanced Control" {
user_input "Location" returns: loc type: point <- {0,0};
create cells number: 10 with: [location::loc];
}
```

- See also: user_command, user_init, user_panel,

Embedments

- The `user_input` statement is of type: **Single statement**
- The `user_input` statement can be embedded into: user_command,
- The `user_input` statement embeds statements:

---

user_panel

Facets

- `name` (an identifier), (omissible) :
- `initial` (boolean):

Definition

It is the basic behavior of the user control architecture (it is similar to state for the FSM architecture). This user_panel translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each user_-panel, like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of user_commands which describe the different action buttons present in the panel.

Usages

- The general syntax is for example:

```
user_panel default initial: true {
    user_input 'Number' returns: number type: int <- 10;
    ask (number among list(cells)){ do die; }
    transition to: "Advanced Control" when: every (10);
}

user_panel "Advanced Control" {
    user_input "Location" returns: loc type: point <- {0,0};
    create cells number: 10 with: [location::loc];
}
```

- See also: user_command, user_init, user_input,

Embedments

- The `user_panel` statement is of type: **Behavior**
- The `user_panel` statement can be embedded into: fsm, user_first, user_last, user_-init, user_only, Species, Experiment, Model,
- The `user_panel` statement embeds statements: user_command,

---

using

Facets

- `topology` (topology), (omissible) : the topology

Definition

`using` is a statement that allows to set the topology to use by its sub-statements. They can gather it by asking the scope to provide it.

Usages

- All the spatial operations are topology-dependent (e.g. neighbors are not the same in a continuous and in a grid topology). So `using` statement allows modelers to specify the topology in which the spatial operation will be computed.

```
float dist <- 0.0;
using topology(grid_ant) {
    d (self.location distance_to target.location);
}
```

Embedments

- The `using` statement is of type: **Sequence of statements or action**
- The `using` statement can be embedded into: chart, Behavior, Sequence of statements or action,
- The `using` statement embeds statements:

Variable_container

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `fill_with` (any type):
- `function` (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- `index` (a datatype identifier):
- `init` (any type): The initial value of the attribute
- `of` (a datatype identifier):
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead

- `size` (any type in [int, point]):
- `type` (a datatype identifier):
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type):

Definition

Allows to declare an attribute of a species or an experiment

Usages

Embedments

- The `Variable_container` statement is of type: **Variable (container)**
- The `Variable_container` statement can be embedded into: Species, Experiment, Model,
- The `Variable_container` statement embeds statements:

Variable_number

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `among` (list): A list of constant values among which the attribute can take its value
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `function` (any type in [int, float]): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- `init` (any type in [int, float]): The initial value of the attribute
- `max` (any type in [int, float]): The maximum value this attribute can take.

- `min` (any type in [int, float]): The minimum value this attribute can take
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `step` (int):
- `type` (a datatype identifier): The type of the attribute, either 'int' or 'float'
- `update` (any type in [int, float]): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type in [int, float]):

Definition

Allows to declare an attribute of a species or experiment

Usages

Embedments

- The `Variable_number` statement is of type: **Variable (number)**
- The `Variable_number` statement can be embedded into: Species, Experiment, Model,
- The `Variable_number` statement embeds statements:

Variable_regular

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `among` (list): A list of constant values among which the attribute can take its value
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `function` (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- `index` (a datatype identifier): The type of the index used to retrieve elements if the type of the attribute is a container type

- `init` (any type): The initial value of the attribute
- `of` (a datatype identifier): The type of the elements contained in the type of this attribute if it is a container type
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `type` (a datatype identifier): The type of this attribute. Can be combined with facets 'of' and 'index' to describe container types
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute
- `value` (any type):

Definition

Allows to declare an attribute of a species or an experiment

Usages

Embedments

- The `Variable_regular` statement is of type: **Variable (regular)**
- The `Variable_regular` statement can be embedded into: Species, Experiment, Model,
- The `Variable_regular` statement embeds statements:

---

warn

Facets

- `message` (string), (omissible) : the message to display as a warning.

Definition

The statement makes the agent output an arbitrary message in the error view as a warning.

Usages

- Emmitting a warning

```
warn 'This is a warning from ' + self;
```

Embedments

- The `warn` statement is of type: **Single statement**
- The `warn` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `warn` statement embeds statements:

---

write

Facets

- `message` (any type), (omissible) : the message to display. Modelers can add some formatting characters to the message (carriage returns, tabs, or Unicode characters), which will be used accordingly in the console.
- `color` (rgb): The color with wich the message will be displayed. Note that different simulations will have different (default) colors to use for this purpose if this facet is not specified

Definition

The statement makes the agent output an arbitrary message in the console.

Usages

- Outputting a message

```
write 'This is a message from ' + self;
```

Embedments

- The `write` statement is of type: **Single statement**
- The `write` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `write` statement embeds statements:

Chapter 64

Types

A variable's or expression's *type* (or *data type*) determines the values it can take, plus the operations that can be performed on or with it. GAML is a statically-typed language, which means that the type of an expression is always known at compile time, and is even enforced with casting operations. There are 4 categories of types:

- primitive types, declared as keyword in the language,
- complex types, also declared as keyword in the language,
- parametric types, a refinement of complex types (mainly children of container) that is dynamically constructed using an enclosing type, a contents type and a key type,
- species types, dynamically constructed from the species declarations made by the modeler (and the built-in species present).

The hierarchy of types in GAML (only primitive and complex types are displayed here, of course, as the other ones are model-dependent) is the following:

Table of contents

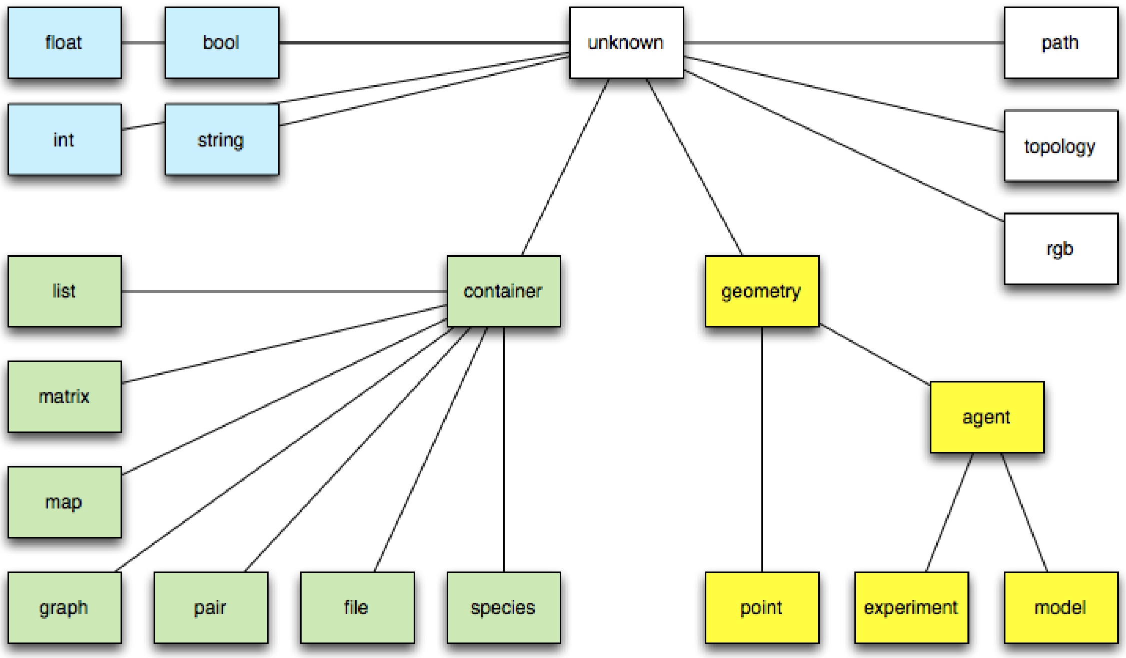


Figure 64.1: images/types_hierarchy.png

Primitive built-in types

bool

- **Definition:** primitive datatype providing two values: `true` or `false`.
- **Litteral declaration:** both `true` or `false` are interpreted as boolean constants.
- **Other declarations:** expressions that require a boolean operand often directly apply a casting to bool to their operand. It is a convenient way to directly obtain a bool value.

```
bool (0) -> false
```

Top of the page

float

- **Definition:** primitive datatype holding floating point values, its absolute value is comprised between 4.9E-324 and 1.8E308.
- **Comments:** this datatype is internally backed up by the Java double datatype.
- **Litteral declaration:** decimal notation 123.45 or exponential notation 123e45 are supported.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to float to their operand. Using it is a way to obtain a float constant.

```
float (12) -> 12.0
```

Top of the page

int

- **Definition:** primitive datatype holding integer values comprised between -2147483648 and 2147483647 (i.e. between `-2^31` and `2^31 - 1`.
- **Comments:** this datatype is internally backed up by the Java int datatype.
- **Litteral declaration:** decimal notation like 1, 256790 or hexadecimal notation like #1209FF are automatically interpreted.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to int to their operand. Using it is a way to obtain an integer constant.

```
int (234.5) -> 234.
```

Top of the page

string

- **Definition:** a datatype holding a sequence of characters.
- **Comments:** this datatype is internally backed up by the Java String class. However, contrary to Java, strings are considered as a primitive type, which means they do not contain character objects. This can be seen when casting a string to a list using the list operator: the result is a list of one-character strings, not a list of characters.

- **Litteral declaration:** a sequence of characters enclosed in quotes, like 'this is a string' . If one wants to literally declare strings that contain quotes, one has to double these quotes in the declaration. Strings accept escape characters like `\n` (newline), `\r` (carriage return), `\t` (tabulation), as well as any Unicode character (`\uXXXX`).
- **Other declarations:** see string
- **Example:** see string operators.

Top of the page

Complex built-in types

Contrarily to primitive built-in types, complex types have often various attributes. They can be accessed in the same way as attributes of agents:

```
complex_type nom_var <- init_var;
ltype_attr attr_var <- nom_var.attr_name;
```

For example:

```
file fileText <- file("../data/cell.Data");
bool fileTextReadable <- fileText.readable;
```

agent

- **Definition:** a generic datatype that represents an agent whatever its actual species.
- **Comments:** This datatype is barely used, since species can be directly used as datatypes themselves.
- **Declaration:** the agent casting operator can be applied to an int (to get the agent with this unique index), a string (to get the agent with this name).

Top of the page

container

- **Definition:** a generic datatype that represents a collection of data.

- **Comments:** a container variable can be a list, a matrix, a map... Conversely each list, matrix and map is a kind of container. In consequence every container can be used in container-related operators.
- **See also:** Container operators
- **Declaration:**

```
container c  <- [1,2,3];
container c  <- matrix [[1,2,3],[4,5,6]];
container c  <- map ["x"::5, "y"::12];
container c  <- list species1;
```

Top of the page

file

- **Definition:** a datatype that represents a file.
- **Built-in attributes:**
  - name (type = string): the name of the represented file (with its extension)
  - extension(type = string): the extension of the file
  - path (type = string): the absolute path of the file
  - readable (type = bool, read-only): a flag expressing whether the file is readable
  - writable (type = bool, read-only): a flag expressing whether the file is writable
  - exists (type = bool, read-only): a flag expressing whether the file exists
  - is_folder (type = bool, read-only): a flag expressing whether the file is folder
  - contents (type = container): a container storing the content of the file
- **Comments:** a variable with the `file` type can handle any kind of file (text, image or shape files...). The type of the `content` attribute will depend on the kind of file. Note that the allowed kinds of file are the followings:
  - text files: files with the extensions .txt, .data, .csv, .text, .tsv, .asc. The `content` is by default a list of string.
  - image files: files with the extensions .pgm, .tif, .tiff, .jpg, .jpeg, .png, .gif, .pict, .bmp. The `content` is by default a matrix of int.
  - shapefiles: files with the extension .shp. The `content` is by default a list of geometry.
  - properties files: files with the extension .properties. The `content` is by default a map of string::string.

  - folders. The `content` is by default a list of string.

- **Remark:** Files are also a particular kind of container and can thus be read, written or iterated using the container operators and commands.
- **See also:** File operators
- **Declaration:** a file can be created using the generic `file` (that opens a file in read only mode and tries to determine its contents), `folder` or the `new_folder` (to open an existing folder or create a new one) unary operators. But things can be specialized with the combination of the `read`/`write` and `image`/`text`/`shapefile`/`properties` unary operators.

```
folder(a_string)  // returns a file managing a existing folder
file(a_string) // returns any kind of file in read-only mode
read(text(a_string)) // returns a text file in read-only mode
read(image(a_string)) // does the same with an image file.
write(properties(a_string)) // returns a property file which is
   available for writing
                            // (if it exists, contents will be
   appended unless it is cleared
                            // using the standard container
   operations).
```

Top of the page

geometry

- **Definition:** a datatype that represents a vector geometry, i.e. a list of georeferenced points.
- **Built-in attributes:**
  - location (type = point): the centroid of the geometry
  - area (type = float): the area of the geometry
  - perimeter (type = float): the perimeter of the geometry
  - holes (type = list of geometry): the list of the hole inside the given geometry
  - contour (type = geometry): the exterior ring of the given geometry and of his holes
  - envelope (type = geometry): the geometry bounding box
  - width (type = float): the width of the bounding box
  - height (type = float): the height of the bounding box

  - points (type = list of point): the set of the points composing the geometry

- **Comments:** a geometry can be either a point, a polyline or a polygon. Operators working on geometries handle transparently these three kinds of geometry. The envelope (a.k.a. the bounding box) of the geometry depends on the kind of geometry:

  - If this Geometry is the empty geometry, it is an empty point.
  - If the Geometry is a point, it is a non-empty point.
  - Otherwise, it is a Polygon whose points are (minx, miny), (maxx, miny), (maxx, maxy), (minx, maxy), (minx, miny).

- **See also:** Spatial operators
- **Declaration:** geometries can be built from a point, a list of points or by using specific operators (circle, square, triangle...).

```
geometry varGeom <- circle(5);
geometry polygonGeom <- polygon([{3,5}, {5,6},{1,4}]);
```

Top of the page

graph

- **Definition:** a datatype that represents a graph composed of vertices linked by edges.
- **Built-in attributes:**

  - edges(type = list of agent/geometry): the list of all edges
  - vertices(type = list of agent/geometry): the list of all vertices
  - circuit (type = path): an approximate minimal traveling salesman tour (hamiltonian cycle)
  - spanning_tree (type = list of agent/geometry): minimum spanning tree of the graph, i.e. a sub-graph such as every vertex lies in the tree, and as much edges lies in it but no cycles (or loops) are formed.
  - connected(type = bool): test whether the graph is connected

- **Remark:**

  - graphs are also a particular kind of container and can thus be manipulated using the container operators and commands.

  - This algorithm used to compute the circuit requires that the graph be complete and the triangle inequality exists (if x,y,z are vertices then d(x,y)+d(y,z)<d(x,z) for all x,y,z) then this algorithm will guarantee a hamiltonian cycle such that the total weight of the cycle is less than or equal to double the total weight of the optimal hamiltonian cycle.
  - The computation of the spanning tree uses an implementation of the Kruskal's minimum spanning tree algorithm. If the given graph is connected it computes the minimum spanning tree, otherwise it computes the minimum spanning forest.

- **See also:** Graph operators
- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create road from: shape_file_road;
graph the_graph <- as_edge_graph(road);

graph([1,9,5])         --: ([1: in[] + out[], 5: in[] + out[], 9:
   in[] + out[]], [])
graph([node(0), node(1), node(2)]       // if node is a species
graph(['a'::345, 'b'::13])  --:  ([b: in[] + out[b::13], a: in[]
   + out[a::345], 13: in[b::13] + out[], 345: in[a::345] + out
   []], [a::345=(a,345), b::13=(b,13)])
graph(a_graph)  --: a_graph
graph(node1)    --: null
```

Top of the page

list

- **Definition:** a composite datatype holding an ordered collection of values.
- **Comments:** lists are more or less equivalent to instances of ArrayList in Java (although they are backed up by a specific class). They grow and shrink as needed, can be accessed via an index (see @ or index_of), support set operations (like union and difference), and provide the modeller with a number of utilities that make it easy to deal with collections of agents (see, for instance, shuffle, reverse,where,sort_by,...).

- **Remark:** lists can contain values of any datatypes, including other lists. Note, however, that due to limitations in the current parser, lists of lists cannot be declared literaly; they have to be built using assignments. Lists are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of expressions separated by commas, enclosed in square brackets, like [12, 14, 'abc', self]. An empty list is noted .
- **Other declarations:** lists can be build litteraly from a point, or a string, or any other element by using the list casting operator.

```
list (1) -> [1]
```

```
list<int> myList <- [1,2,3,4];
myList[2] => 3
```

Top of the page

map

- **Definition:** a composite datatype holding an ordered collection of pairs (a key, and its associated value).
- **Built-in attributes:**
  - keys (type = list): the list of all keys
  - values (type = list): the list of all values
  - pairs (type = list of pairs): the list of all pairs key::value
- **Comments:** maps are more or less equivalent to instances of Hashtable in Java (although they are backed up by a specific class).
- **Remark:** maps can contain values of any datatypes, including other maps or lists. Maps are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of pair expressions separated by commas, enclosed in square brackets; each pair is represented by a key and a value sperarated by '::'. An example of map is [agentA::'big', agentB::'small', agentC::'big']. An empty map is noted .
- **Other declarations:** lists can be built litteraly from a point, or a string, or any other element by using the map casting operator.

```
map (1) -> [1::1]
map ({1,5}) -> [x::1, y::5]
[]   // empty map
```

Top of the page

matrix

- **Definition:** a composite datatype that represents either a two-dimension array (matrix) or a one-dimension array (vector), holding any type of data (including other matrices).
- **Comments:** Matrices are fixed-size structures that can be accessed by index (point for two-dimensions matrices, integer for vectors).
- **Litteral declaration:** Matrices cannot be defined literally. One-dimensions matrices can be built by using the matrix casting operator applied on a list. Two-dimensions matrices need to to be declared as variables first, before being filled.

```
//builds a one-dimension matrix, of size 5
matrix mat1 <- matrix ([10, 20, 30, 40, 50]);
//  builds a two-dimensions matrix with 10 columns and 5 rows,
   where each cell is initialized to 0.0
matrix mat2 <- 0.0 as_matrix({10,5});
// builds a two-dimensions matrix with 2 columns and 3 rows, with
    initialized cells
matrix mat3 <- matrix([["c11","c12","c13"],["c21","c22","c23"]]);
    -> c11;c21
       c12;c22
       c13;c23
```

Top of the page

pair

- **Definition:** a datatype holding a key and its associated value.
- **Built-in attributes:**
    - key (type = string): the key of the pair, i.e. the first element of the pair

  - value (type = string): the value of the pair, i.e. the second element of the pair

- **Remark:** pairs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a pair is defined by a key and a value sperarated by '::'.
- **Other declarations:** a pair can also be built from:

  - a point,
  - a map (in this case the first element of the pair is the list of all the keys of the map and the second element is the list of all the values of the map),
  - a list (in this case the two first element of the list are used to built the pair)

```
pair testPair <- "key"::56;
pair testPairPoint <- {3,5};             // 3::5
pair testPairList2 <- [6,7,8];           // 6::7
pair testPairMap <- [2::6,5::8,12::45];  // [12,5,2]::[45,8,6]
```

Top of the page

path

- **Definition:** a datatype representing a path linking two agents or geometries in a graph.
- **Built-in attributes:**

  - source (type = point): the source point, i.e. the first point of the path
  - target (type = point): the target point, i.e. the last point of the path
  - graph (type = graph): the current topology (in the case it is a spatial graph), null otherwise
  - edges (type = list of agents/geometries) : the edges of the graph composing the path
  - vertices (type = list of agents/geometries) : the vertices of the graph composing the path
  - segments (type = list of geometries): the list of the geometries composing the path
  - shape (type = geometry) : the global geometry of the path (polyline)

- **Comments:** the path created between two agents/geometries or locations will strongly depends on the topology in which it is created.

- **Remark:** a path is **immutable**, i.e. it can not be modified after it is created.
- **Declaration:** paths are very barely defined litterally. We can nevertheless use the `path` unary operator on a list of points to build a path. Operators dedicated to the computation of paths (such as path_to or path_between) are often used to build a path.

```
path([{1,5},{2,9},{5,8}]) // a path from {1,5} to {5,8} through
   {2,9}

geometry rect <- rectangle(5);
geometry poly <- polygon([{10,20},{11,21},{10,21},{11,22}]);
path pa <- rect path_to poly;  // built a path between rect and
   poly, in the topology
                                             // of the current
   agent (i.e. a line in a& continuous topology,
                                             // a path in a graph
    in a graph topology )

a_topology path_between a_container_of_geometries // idem with an
    explicit topology and the possiblity
                                                   // to have more
    than 2 geometries
                                                   // (the path is
    then built incrementally)


path_between (a_graph, a_source, a_target) // idem with a the
   given graph as topology
```

Top of the page

point

- **Definition:** a datatype normally holding two positive float values. Represents the absolute coordinates of agents in the model.
- **Built-in attributes:**
  - x (type = float): coordinate of the point on the x-axis
  - y (type = float): coordinate of the point on the y-axis

- **Comments:** point coordinates should be positive, if a negative value is used in its declaration, the point is built with the absolute value.
- **Remark:** points are particular cases of geometries and containers. Thus they have also all the built-in attributes of both the geometry and the container datatypes and can be used with every kind of operator or command admitting geometry and container.
- **Litteral declaration:** two numbers, separated by a comma, enclosed in braces, like {12.3, 14.5}
- **Other declarations:** points can be built litteraly from a list, or from an integer or float value by using the point casting operator.

```
point ([12,123.45]) -> {12.0, 123.45}
point (2) -> {2.0, 2.0}
```

Top of the page

rgb

- **Definition:** a datatype that represents a color in the RGB space.
- **Built-in attributes:**
  - red(type = int): the red component of the color
  - green(type = int): the green component of the color
  - blue(type = int): the blue component of the color
  - darker(type = rgb): a new color that is a darker version of this color
  - brighter(type = rgb): a new color that is a brighter version of this color
- **Remark:** rgbs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** there exist lot of ways to declare a color. We use the `rgb` casting operator applied to:
  - a string. The allowed color names are the constants defined in the Color Java class, i.e.: black, blue, cyan, darkGray, lightGray, gray, green, magenta, orange, pink, red, white, yellow.
  - a list. The integer value associated to the three first elements of the list are used to define the three red (element 0 of the list), green (element 1 of the list) and blue (element 2 of the list) components of the color.

  - a map. The red, green, blue compoenents take the value associated to the keys "r", "g", "b" in the map.
  - an integer <- the decimal integer is translated into a hexadecimal <- OxRRGGBB. The red (resp. green, blue) component of the color take the value RR (resp. GG, BB) translated in decimal.
  - Since GAMA 1.6.1, colors can be directly obtained like units, by using the ° or # symbol followed by the name in lowercase of one of the 147 CSS colors (see http://www.cssportal.com/css3-color-names/).

- **Declaration:**

```
rgb cssRed <- #red;   // Since 1.6.1
rgb testColor <- rgb('white');                    // rgb
   [255,255,255]
rgb test <- rgb(3,5,67);                         // rgb [3,5,67]
rgb te <- rgb(340);                               // rgb [0,1,84]
rgb tete <- rgb(["r"::34, "g"::56, "b"::345]); // rgb [34,56,255]
```

Top of the page

species

- Definition: a generic datatype that represents a species
- **Built-in attributes:**

  - topology (type=topology): the topology is which lives the population of agents

- Comments: this datatype is actually a "meta-type". It allows to manipulate (in a rather limited fashion, however) the species themselves as any other values.
- Litteral declaration: the name of a declared species is already a litteral declaration of species.
- Other declarations: the species casting operator, or its variant called species_of can be applied to an agent in order to get its species.

Top of the page

Species names as types

Once a species has been declared in a model, it automatically becomes a datatype. This means that : * It can be used to declare variables, parameters or constants, * It can be used as an operand to commands or operators that require species parameters, * It can be used as a casting operator (with the same capabilities as the built-in type agent)

In the simple following example, we create a set of "humans" and initialize a random "friendship network" among them. See how the name of the species, human, is used in the create command, as an argument to the list casting operator, and as the type of the variable named friend.

```
global {
    init {
         create human number: 10;
         ask human {
               friend <- one_of (human - self);
         }
     }
}
entities {
    species human {
        human friend <- nil;
    }
}
```

Top of the page

topology

- **Definition:** a topology is basically on neighborhoods, distance,... structures in which agents evolves. It is the environment or the context in which all these values are computed. It also provides the access to the spatial index shared by all the agents. And it maintains a (eventually dynamic) link with the 'environment' which is a geometrical border.
- **Built-in attributes:**
  - places(type = container): the collection of places (geometry) defined by this topology.

    - environment(type = geometry): the environment of this topology (i.e. the geometry that defines its boundaries)

- **Comments:** the attributes `places` depends on the kind of the considered topolopy. For continuous topologies, it is a list with their environment. For discrete topologies, it can be any of the container supporting the inclusion of geometries (list, graph, map, matrix)
- **Remark:** There exist various kinds of topology: continuous topology and discrete topology (e.g. grid, graph…)
- **Declaration:** To create a topology, we can use the `topology` unary casting operator applied to:

    - an agent: returns a continuous topology built from the agent's geometry
    - a species name: returns the topology defined for this species population
    - a geometry: returns a continuous topology built on this geometry
    - a geometry container (list, map, shapefile): returns an half-discrete (with corresponding places), half-continuous topology (to compute distances…)
    - a geometry matrix (i.e. a grid): returns a grid topology which computes specifically neighborhood and distances
    - a geometry graph: returns a graph topology which computes specifically neighborhood and distances More complex topologies can also be built using dedicated operators, e.g. to decompose a geometry…

Top of the page

Defining custom types

Sometimes, besides the species of agents that compose the model, it can be necessary to declare custom datatypes. Species serve this purpose as well, and can be seen as "classes" that can help to instantiate simple "objects". In the following example, we declare a new kind of "object", bottle, that lacks the skills habitually associated with agents (moving, visible, etc.), but can nevertheless group together attributes and behaviors within the same closure. The following example demonstrates how to create the species:

```
species bottle {
    float volume <- 0.0 max:1 min:0.0;
    bool is_empty -> {volume = 0.0};
    action fill {
        volume <- 1.0;
    }
}
```

How to use this species to declare new bottles :

```
create bottle {
    volume <- 0.5;
}
```

And how to use bottles as any other agent in a species (a drinker owns a bottle; when he gets thirsty, it drinks a random quantity from it; when it is empty, it refills it):

```
species drinker {
     ...
    bottle my_bottle<- nil;
    float quantity <- rnd (100) / 100;
    bool thirsty <- false update: flip (0.1);
    ...
    action drink {
         if condition: ! bottle.is_empty {
              bottle.volume <-bottle.volume - quantity;
              thirsty <- false;
         }
    }
    ...
    init {
          create bottle return: created_bottle;
              volume <- 0.5;
          }
          my_bottle <- first(created_bottle);
    }
    ...
    reflex filling_bottle when: bottle.is_empty {
         ask  my_bottle {
              do fill;
         }
    }
    ...
    reflex drinking when: thirsty {
         do drink;
```

```
      }
}
```

Top of the page

Chapter 65

File Types

GAMA provides modelers with a generic type for files called **file**. It is possible to load a file using the *file* operator:

```
file my_file <- file("../includes/data.csv");
```

However, internally, GAMA makes the difference between the different types of files. Indeed, for instance:

```
global {
    init {
        file my_file <- file("../includes/data.csv");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
```

```
0.2
Iris-setosa
4.9
3.0
1.4
0.2
Iris-setosa
...
```

Indeed, the content of CSV file is a matrix: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default “,”).

In contrary:

```
global {
    init {
        file my_file <- file("../includes/data.shp");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
```

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile.

In order to know how to load a file, GAMA analyzes its extension. For instance for a file with a “.csv” extension, GAMA knows that the file is a **csv** one and will try to split each line with the *,* separator. However, if the modeler wants to split each line with a different separator (for instance **;**) or load it as a text file, he/she will have to use a specific file operator.

Indeed, GAMA integrates specific operators corresponding to different types of files.

Table of contents

Text File

Extensions

Here the list of possible extensions for text file: * "txt" * "data" * "csv" * "text" * "tsv" * "xml"

Note that when trying to define the type of a file with the default file loading operator (**file**), GAMA will first try to test the other type of file. For example, for files with ".csv" extension, GAMA will cast them as csv file and not as text file.

Content

The content of a text file is a list of string corresponding to each line of the text file. For example:

```
global {
    init {
        file my_file <- text_file("../includes/data.txt");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength,sepalwidth,petallength,petalwidth,type
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
```

Operators

List of operators related to text files: * **text_file(string path)**: load a file (with an authorized extension) as a text file. * **text_file(string path, list content)**: load a file (with an authorized extension) as a text file and fill it with the given content. * **is_text(op)**: tests whether the operand is a text file

CSV File

Extensions

Here the list of possible extensions for csv file: * "csv" * "tsv"

Content

The content of a csv file is a matrix of string: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default ","). For example:

```
global {
    init {
        file my_file <- csv_file("../includes/data.csv");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
0.2
Iris-setosa
4.9
3.0
1.4
0.2
Iris-setosa
...
```

Operators

List of operators related to csv files: * **csv_file(string path)**: load a file (with an authorized extension) as a csv file with default separator (","). * **csv_file(string path, string separator)**: load a file (with an authorized extension) as a csv file with the given separator.

```
file my_file <- csv_file("../includes/data.csv", ";");
```

- **csv_file(string path, matrix content)**: load a file (with an authorized extension) as a csv file and fill it with the given content.
- **is_csv(op)**: tests whether the operand is a csv file

Shapefile

Shapefiles are classical GIS data files. A shapefile is not simple file, but a set of several files (source: wikipedia): * Mandatory files : * .shp — shape format; the feature geometry itself * .shx — shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly * .dbf — attribute format; columnar attributes for each shape, in dBase IV format

- Optional files :
  - .prj — projection format; the coordinate system and projection information, a plain text file describing the projection using well-known text format
  - .sbn and .sbx — a spatial index of the features
  - .fbn and .fbx — a spatial index of the features for shapefiles that are read-only
  - .ain and .aih — an attribute index of the active fields in a table
  - .ixs — a geocoding index for read-write shapefiles
  - .mxs — a geocoding index for read-write shapefiles (ODB format)
  - .atx — an attribute index for the .dbf file in the form of shapefile.columnname.atx (ArcGIS 8 and later)
  - .shp.xml — geospatial metadata in XML format, such as ISO 19115 or other XML schema
  - .cpg — used to specify the code page (only for .dbf) for identifying the character encoding to be used

More details about shapefiles can be found here.

Extensions

Here the list of possible extension for shapefile: * "shp"

Content

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile. For example:

```
global {
    init {
        file my_file <- shape_file("../includes/data.shp");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
...
```

Note that the attributes of each object of the shapefile is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of a corresponding attributes.

For example:

```
file my_file <- shape_file("../includes/data.shp");
write "my_file: " + my_file.contents;
loop el over: my_file {
    write (el get "TYPE");
}
```

Operators

List of operators related to shapefiles: * **shape_file(string path)**: load a file (with an authorized extension) as a shapefile with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). * **shape_file(string path, string code)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/) * **shape_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/)

```
file my_file <- shape_file("../includes/data.shp", "EPSG:32601");
```

- **shape_file(string path, list content)**: load a file (with an authorized extension) as a shapefile and fill it with the given content.
- **is_shape(op)**: tests whether the operand is a shapefile

OSM File

OSM (Open Street Map) is a collaborative project to create a free editable map of the world. The data produced in this project (OSM File) represent physical features on the ground (e.g., roads or buildings) using tags attached to its basic data structures (its nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way or relation (source: openstreetmap.org).

More details about OSM data can be found here.

Extensions

Here the list of possible extension for shapefile: * "osm" * "pbf" * "bz2" * "gz"

Content

The content of a OSM data is a list of geometries corresponding to the objects of the OSM file. For example:

```
global {
    init {
        file my_file <- osm_file("../includes/data.gz");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Point
Point
Point
Point
Point
LineString
LineString
Polygon
Polygon
```

```
Polygon
...
```

Note that like for shapefiles, the attributes of each object of the osm file is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of a corresponding attributes.

Operators

List of operators related to osm file: * **osm_file(string path)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, string code)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry.

```
file my_file <- osm_file("../includes/data.gz", "EPSG:32601");
```

- **osm_file(string path, map filter)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, only the elements with the defined values are loaded from the file. "' //map used to filter the object to build from the OSM file according to attributes. map filtering <- map(["highway"::["primary", "secondary", "tertiary", "motorway", "living_street","residential", "unclassified"], "building"::["yes"]]);

//OSM file to load file osmfile <- file

Constants

- `#e`, value= 2.718281828459045, Comment: The e constant
- `#infinity`, value= Infinity, Comment: A constant holding the positive infinity of type (Java Double.POSITIVE_INFINITY)
- `#max_float`, value= 1.7976931348623157E308, Comment: A constant holding the largest positive finite value of type float (Java Double.MAX_VALUE)
- `#max_int`, value= 2.147483647E9, Comment: A constant holding the maximum value an int can have (Java Integer.MAX_VALUE)
- `#min_float`, value= 4.9E-324, Comment: A constant holding the smallest positive nonzero value of type float (Java Double.MIN_VALUE)
- `#min_int`, value= -2.147483648E9, Comment: A constant holding the minimum value an int can have (Java Integer.MIN_VALUE)
- `#nan`, value= NaN, Comment: A constant holding a Not-a-Number (NaN) value of type float (Java Double.POSITIVE_INFINITY)
- `#pi`, value= 3.141592653589793, Comment: The PI constant
- `#to_deg`, value= 57.29577951308232, Comment: A constant holding the value to convert radians into degrees
- `#to_rad`, value= 0.017453292519943295, Comment: A constant holding the value to convert degrees into radians

---

Graphics units

- `#bold`, value= 1, Comment: This constant allows to build a font with a bold face. Can be combined with #italic
- `#camera_location`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current position of the camera as a point
- `#camera_orientation`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current orientation of the camera as a point
- `#camera_target`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current target of the camera as a point
- `#display_height`, value= 1.0, Comment: This constant is only accessible in a graphical context: display, graphics…
- `#display_width`, value= 1.0, Comment: This constant is only accessible in a graphical context: display, graphics…
- `#flat`, value= 2, Comment: This constant represents a flat line buffer end cap style
- `#horizontal`, value= 3, Comment: This constant represents a layout where all display views are aligned horizontally
- `#italic`, value= 2, Comment: This constant allows to build a font with an italic face. Can be combined with #bold
- `#none`, value= 0, Comment: This constant represents the absence of a predefined layout
- `#pixels` (#px), value= 1.0, Comment: This unit, only available when running aspects or declaring displays, returns a dynamic value instead of a fixed one. px (or pixels), returns the value of one pixel on the current view in terms of model units.

- **#plain**, value= 0, Comment: This constant allows to build a font with a plain face

- **#round**, value= 1, Comment: This constant represents a round line buffer end cap style

- **#split**, value= 2, Comment: This constant represents a layout where all display views are split in a grid-like structure

- **#square**, value= 3, Comment: This constant represents a square line buffer end cap style

- **#stack**, value= 1, Comment: This constant represents a layout where all display views are stacked

- **#user_location**, value= No Default Value, Comment: This unit contains in permanence the location of the mouse on the display in which it is situated. The latest location is provided when it is out of a display

- **#vertical**, value= 4, Comment: This constant represents a layout where all display views are aligned vertically

- **#zoom**, value= 1.0, Comment: This unit, only available when running aspects or declaring displays, returns the current zoom level of the display as a positive float, where 1.0 represent the neutral zoom (100%)

Length units

- **#cm** (#centimeter,#centimeters), value= 0.009999999776482582, Comment: centimeter unit

- **#dm** (#decimeter,#decimeters), value= 0.10000000149011612, Comment: decimeter unit

- **`#foot`** (#feet,#ft), value= 0.3047999931871891, Comment: foot unit
- **`#inch`** (#inches), value= 0.025399999432265757, Comment: inch unit
- **`#km`** (#kilometer,#kilometers), value= 1000.0, Comment: kilometer unit
- **`#m`** (#meter,#meters), value= 1.0, Comment: meter: the length basic unit
- **`#mile`** (#miles), value= 1609.344, Comment: mile unit
- **`#mm`** (#milimeter,#milimeters), value= 9.999999776482583E-4, Comment: millimeter unit
- **`#yard`** (#yards), value= 0.9144, Comment: yard unit

Surface units

- **`#m2`**, value= 1.0, Comment: square meter: the basic unit for surfaces
- **`#sqft`** (#square_foot,#square_feet), value= 0.09290303584691051, Comment: square foot unit
- **`#sqin`** (#square_inch,#square_inches), value= 6.451599711591008E-4, Comment: square inch unit
- **`#sqmi`** (#square_mile,#square_miles), value= 2589988.110336, Comment: square mile unit

---

Time units

- **`#day`** (#days,#day), value= 86400.0, Comment: day time unit
- **`#h`** (#hour,#hours), value= 3600.0, Comment: hour time unit
- **`#minute`** (#minutes,#mn), value= 60.0, Comment: minute time unit
- **`#month`** (#months), value= 2592000.0, Comment: month time unit. Note that 1 month equals 30 days and 1 year 360 days in these units
- **`#msec`** (#millisecond,#milliseconds,#ms), value= 0.001, Comment: millisecond time unit
- **`#now`**, value= 1.0, Comment: This constant represents the current date
- **`#sec`** (#second,#seconds,#s), value= 1.0, Comment: second: the time basic unit
- **`#year`** (#years,#y), value= 3.1104E7, Comment: year time unit. Note that 1 month equals 30 days and 1 year 360 days in these units

---

Volume units

- **`#cl`** (#centiliter,#centiliters), value= 1.0E-5, Comment: centiliter unit
- **`#dl`** (#deciliter,#deciliters), value= 1.0E-4, Comment: deciliter unit
- **`#hl`** (#hectoliter,#hectoliters), value= 0.1, Comment: hectoliter unit
- **`#l`** (#liter,#liters,#dm3), value= 0.001, Comment: liter unit
- **`#m3`**, value= 1.0, Comment: cube meter: the basic unit for volumes

---

Weight units

- **`#gram`** (#grams), value= 0.001, Comment: gram unit
- **`#kg`** (#kilo,#kilogram,#kilos), value= 1.0, Comment: second: the basic unit for weights
- **`#longton`** (#lton), value= 1016.0469088000001, Comment: short ton unit
- **`#ounce`** (#oz,#ounces), value= 0.028349523125, Comment: ounce unit
- **`#pound`** (#lb,#pounds,#lbm), value= 0.45359237, Comment: pound unit
- **`#shortton`** (#ston), value= 907.18474, Comment: short ton unit
- **`#stone`** (#st), value= 6.35029318, Comment: stone unit
- **`#ton`** (#tons), value= 1000.0, Comment: ton unit

Colors

In addition to the previous units, GAML provides a direct access to the 147 named colors defined in CSS (see http://www.cssportal.com/css3-color-names/). E.g,

```
rgb my_color <- °teal;
```

- **`#aliceblue`**, value= r=240, g=248, b=255, alpha=1
- **`#antiquewhite`**, value= r=250, g=235, b=215, alpha=1
- **`#aqua`**, value= r=0, g=255, b=255, alpha=1

- `#aquamarine`, value= r=127, g=255, b=212, alpha=1
- `#azure`, value= r=240, g=255, b=255, alpha=1
- `#beige`, value= r=245, g=245, b=220, alpha=1
- `#bisque`, value= r=255, g=228, b=196, alpha=1
- `#black`, value= r=0, g=0, b=0, alpha=1
- `#blanchedalmond`, value= r=255, g=235, b=205, alpha=1
- `#blue`, value= r=0, g=0, b=255, alpha=1
- `#blueviolet`, value= r=138, g=43, b=226, alpha=1
- `#brown`, value= r=165, g=42, b=42, alpha=1
- `#burlywood`, value= r=222, g=184, b=135, alpha=1
- `#cadetblue`, value= r=95, g=158, b=160, alpha=1
- `#chartreuse`, value= r=127, g=255, b=0, alpha=1
- `#chocolate`, value= r=210, g=105, b=30, alpha=1
- `#coral`, value= r=255, g=127, b=80, alpha=1
- `#cornflowerblue`, value= r=100, g=149, b=237, alpha=1
- `#cornsilk`, value= r=255, g=248, b=220, alpha=1
- `#crimson`, value= r=220, g=20, b=60, alpha=1
- `#cyan`, value= r=0, g=255, b=255, alpha=1

- `#darkblue`, value= r=0, g=0, b=139, alpha=1
- `#darkcyan`, value= r=0, g=139, b=139, alpha=1
- `#darkgoldenrod`, value= r=184, g=134, b=11, alpha=1
- `#darkgray`, value= r=169, g=169, b=169, alpha=1
- `#darkgreen`, value= r=0, g=100, b=0, alpha=1
- `#darkgrey`, value= r=169, g=169, b=169, alpha=1
- `#darkkhaki`, value= r=189, g=183, b=107, alpha=1
- `#darkmagenta`, value= r=139, g=0, b=139, alpha=1
- `#darkolivegreen`, value= r=85, g=107, b=47, alpha=1
- `#darkorange`, value= r=255, g=140, b=0, alpha=1
- `#darkorchid`, value= r=153, g=50, b=204, alpha=1
- `#darkred`, value= r=139, g=0, b=0, alpha=1
- `#darksalmon`, value= r=233, g=150, b=122, alpha=1
- `#darkseagreen`, value= r=143, g=188, b=143, alpha=1
- `#darkslateblue`, value= r=72, g=61, b=139, alpha=1
- `#darkslategray`, value= r=47, g=79, b=79, alpha=1
- `#darkslategrey`, value= r=47, g=79, b=79, alpha=1
- `#darkturquoise`, value= r=0, g=206, b=209, alpha=1

- `#darkviolet`, value= r=148, g=0, b=211, alpha=1
- `#deeppink`, value= r=255, g=20, b=147, alpha=1
- `#deepskyblue`, value= r=0, g=191, b=255, alpha=1
- `#dimgray`, value= r=105, g=105, b=105, alpha=1
- `#dimgrey`, value= r=105, g=105, b=105, alpha=1
- `#dodgerblue`, value= r=30, g=144, b=255, alpha=1
- `#firebrick`, value= r=178, g=34, b=34, alpha=1
- `#floralwhite`, value= r=255, g=250, b=240, alpha=1
- `#forestgreen`, value= r=34, g=139, b=34, alpha=1
- `#fuchsia`, value= r=255, g=0, b=255, alpha=1
- `#gainsboro`, value= r=220, g=220, b=220, alpha=1
- `#ghostwhite`, value= r=248, g=248, b=255, alpha=1
- `#gold`, value= r=255, g=215, b=0, alpha=1
- `#goldenrod`, value= r=218, g=165, b=32, alpha=1
- `#gray`, value= r=128, g=128, b=128, alpha=1
- `#green`, value= r=0, g=128, b=0, alpha=1
- `#greenyellow`, value= r=173, g=255, b=47, alpha=1
- `#grey`, value= r=128, g=128, b=128, alpha=1

- `#honeydew`, value= r=240, g=255, b=240, alpha=1
- `#hotpink`, value= r=255, g=105, b=180, alpha=1
- `#indianred`, value= r=205, g=92, b=92, alpha=1
- `#indigo`, value= r=75, g=0, b=130, alpha=1
- `#ivory`, value= r=255, g=255, b=240, alpha=1
- `#khaki`, value= r=240, g=230, b=140, alpha=1
- `#lavender`, value= r=230, g=230, b=250, alpha=1
- `#lavenderblush`, value= r=255, g=240, b=245, alpha=1
- `#lawngreen`, value= r=124, g=252, b=0, alpha=1
- `#lemonchiffon`, value= r=255, g=250, b=205, alpha=1
- `#lightblue`, value= r=173, g=216, b=230, alpha=1
- `#lightcoral`, value= r=240, g=128, b=128, alpha=1
- `#lightcyan`, value= r=224, g=255, b=255, alpha=1
- `#lightgoldenrodyellow`, value= r=250, g=250, b=210, alpha=1
- `#lightgray`, value= r=211, g=211, b=211, alpha=1
- `#lightgreen`, value= r=144, g=238, b=144, alpha=1
- `#lightgrey`, value= r=211, g=211, b=211, alpha=1
- `#lightpink`, value= r=255, g=182, b=193, alpha=1

- **`#lightsalmon`**, value= r=255, g=160, b=122, alpha=1
- **`#lightseagreen`**, value= r=32, g=178, b=170, alpha=1
- **`#lightskyblue`**, value= r=135, g=206, b=250, alpha=1
- **`#lightslategray`**, value= r=119, g=136, b=153, alpha=1
- **`#lightslategrey`**, value= r=119, g=136, b=153, alpha=1
- **`#lightsteelblue`**, value= r=176, g=196, b=222, alpha=1
- **`#lightyellow`**, value= r=255, g=255, b=224, alpha=1
- **`#lime`**, value= r=0, g=255, b=0, alpha=1
- **`#limegreen`**, value= r=50, g=205, b=50, alpha=1
- **`#linen`**, value= r=250, g=240, b=230, alpha=1
- **`#magenta`**, value= r=255, g=0, b=255, alpha=1
- **`#maroon`**, value= r=128, g=0, b=0, alpha=1
- **`#mediumaquamarine`**, value= r=102, g=205, b=170, alpha=1
- **`#mediumblue`**, value= r=0, g=0, b=205, alpha=1
- **`#mediumorchid`**, value= r=186, g=85, b=211, alpha=1
- **`#mediumpurple`**, value= r=147, g=112, b=219, alpha=1
- **`#mediumseagreen`**, value= r=60, g=179, b=113, alpha=1
- **`#mediumslateblue`**, value= r=123, g=104, b=238, alpha=1

- `#mediumspringgreen`, value= r=0, g=250, b=154, alpha=1
- `#mediumturquoise`, value= r=72, g=209, b=204, alpha=1
- `#mediumvioletred`, value= r=199, g=21, b=133, alpha=1
- `#midnightblue`, value= r=25, g=25, b=112, alpha=1
- `#mintcream`, value= r=245, g=255, b=250, alpha=1
- `#mistyrose`, value= r=255, g=228, b=225, alpha=1
- `#moccasin`, value= r=255, g=228, b=181, alpha=1
- `#navajowhite`, value= r=255, g=222, b=173, alpha=1
- `#navy`, value= r=0, g=0, b=128, alpha=1
- `#oldlace`, value= r=253, g=245, b=230, alpha=1
- `#olive`, value= r=128, g=128, b=0, alpha=1
- `#olivedrab`, value= r=107, g=142, b=35, alpha=1
- `#orange`, value= r=255, g=165, b=0, alpha=1
- `#orangered`, value= r=255, g=69, b=0, alpha=1
- `#orchid`, value= r=218, g=112, b=214, alpha=1
- `#palegoldenrod`, value= r=238, g=232, b=170, alpha=1
- `#palegreen`, value= r=152, g=251, b=152, alpha=1
- `#paleturquoise`, value= r=175, g=238, b=238, alpha=1

- **`#palevioletred`**, value= r=219, g=112, b=147, alpha=1
- **`#papayawhip`**, value= r=255, g=239, b=213, alpha=1
- **`#peachpuff`**, value= r=255, g=218, b=185, alpha=1
- **`#peru`**, value= r=205, g=133, b=63, alpha=1
- **`#pink`**, value= r=255, g=192, b=203, alpha=1
- **`#plum`**, value= r=221, g=160, b=221, alpha=1
- **`#powderblue`**, value= r=176, g=224, b=230, alpha=1
- **`#purple`**, value= r=128, g=0, b=128, alpha=1
- **`#red`**, value= r=255, g=0, b=0, alpha=1
- **`#rosybrown`**, value= r=188, g=143, b=143, alpha=1
- **`#royalblue`**, value= r=65, g=105, b=225, alpha=1
- **`#saddlebrown`**, value= r=139, g=69, b=19, alpha=1
- **`#salmon`**, value= r=250, g=128, b=114, alpha=1
- **`#sandybrown`**, value= r=244, g=164, b=96, alpha=1
- **`#seagreen`**, value= r=46, g=139, b=87, alpha=1
- **`#seashell`**, value= r=255, g=245, b=238, alpha=1
- **`#sienna`**, value= r=160, g=82, b=45, alpha=1
- **`#silver`**, value= r=192, g=192, b=192, alpha=1

- **#skyblue**, value= r=135, g=206, b=235, alpha=1
- **#slateblue**, value= r=106, g=90, b=205, alpha=1
- **#slategray**, value= r=112, g=128, b=144, alpha=1
- **#slategrey**, value= r=112, g=128, b=144, alpha=1
- **#snow**, value= r=255, g=250, b=250, alpha=1
- **#springgreen**, value= r=0, g=255, b=127, alpha=1
- **#steelblue**, value= r=70, g=130, b=180, alpha=1
- **#tan**, value= r=210, g=180, b=140, alpha=1
- **#teal**, value= r=0, g=128, b=128, alpha=1
- **#thistle**, value= r=216, g=191, b=216, alpha=1
- **#tomato**, value= r=255, g=99, b=71, alpha=1
- **#transparent**, value= r=0, g=0, b=0, alpha=0
- **#turquoise**, value= r=64, g=224, b=208, alpha=1
- **#violet**, value= r=238, g=130, b=238, alpha=1
- **#wheat**, value= r=245, g=222, b=179, alpha=1
- **#white**, value= r=255, g=255, b=255, alpha=1
- **#whitesmoke**, value= r=245, g=245, b=245, alpha=1
- **#yellow**, value= r=255, g=255, b=0, alpha=1
- **#yellowgreen**, value= r=154, g=205, b=50, alpha=1

Chapter 66

Pseudo-variables

The expressions known as **pseudo-variables** are special read-only variables that are not declared anywhere (at least not in a species), and which represent a value that changes depending on the context of execution.

Table of contents

self

The pseudo-variable `self` always holds a reference to the agent executing the current statement.

- Example (sets the `friend` attribute of another random agent of the same species to `self` and conversely):

```
friend potential_friend <- one_of (species(self) - self);
if potential_friend != nil {
    potential_friend.friend <- self;
    friend <- potential_friend;
}
```

myself

`myself` plays the same role as `self` but in remotely-executed code (`ask`, `create`, `capture` and `release` statements), where it represents the *calling* agent when the code is executed by the *remote* agent.

- Example (asks the first agent of my species to set its color to my color):

```
ask first (species (self)){
    color <- myself.color;
}
```

- Example (create 10 new agents of the species of my species, share the energy between them, turn them towards me, and make them move 4 times to get closer to me):

```
create species (self) number: 10 {
   energy <- myself.energy / 10.0;
   loop times: 4 {
       heading <- towards (myself);
       do move;
   }
}
```

each

`each` is available only in the right-hand argument of iterators. It is a pseudo-variable that represents, in turn, each of the elements of the left-hand container. It can then take any type depending on the context.

- Example:

```
 list<string> names <- my_species collect each.name;  // each
is of type my_species
 int max <- max(['aa', 'bbb', 'cccc'] collect length(each));
// each is of type string
```

Chapter 67

Variables and Attributes

Variables and attributes represent named data that can be used in an expression. They can be accessed depending on their *scope*: * the scope of attributes declared in a species is itself, its child species and its micro-species. * the scope of temporary variables is the one in which they have been declared, and all its sub-scopes. Outside its *scope* of validity, an expression cannot use a variable or an attribute directly. However, attributes can be used in a remote fashion by using a dotted notation on a given agent (see here).

Table of contents

Direct Access

When an agent wants to use either one of the variables declared locally, one of the attributes declared in its species (or parent species), one of the attributes declared in the macro-species of its species, it can directly invoke its name and the compiler will do the rest (i.e. finding the variable or attribute in the right scope). For instance, we can have a look at the following example:

```
species animal {
   float energy <- 1000 min: 0 max: 2000 update: energy - 0.001;
   int age_in_years <- 1 update: age_in_years + int (time / 365);

   action eat (float amount <- 0) {
       float gain <- amount / age_in_years;
       energy <- energy + gain;
   }

   reflex feed {
      int food_found <- rnd(100);
      do eat (amount: food_found);
   }

}
```

- **Species declaration** > Everywhere in the species declaration, we are able to directly name and use:
  - time, a global built-in variable,
  - energy and age_in_years, the two species attributes. > Nevertheless, in the species declaration, but outside of the action eat and the reflex feed, we **cannot** name the variables:
  - amount, the argument of eat action,
  - gain, a local variable defined into the eat action,
  - food_found, the local variable defined into the feed reflex.
- **Eat action declaration** > In the eat action declaration, we can directly name and use:
  - time, a global built-in variable,
  - energy and age_in_years, the two species attributes,
  - amount, which is an argument to the action eat,
  - gain, a temporary variable within the action. > We **cannot** name and use the variables:
  - food_found, the local variable defined into the feed reflex.
- **feed reflex declaration** > Similarly, in the feed reflex declaration, we can directly name and use:

- `time`, a global built-in variable,
- `energy` and `age_in_years`, the two species variables,
- `food_found`, the local variable defined into the reflex. > But we **cannot** access to variables:
- `amount`, the argument of `eat` action,
- `gain`, a local variable defined into the `eat` action.

Remote Access

When an expression needs to get access to the attribute of an agent which does not belong to its scope of execution, a special notation (similar to that used in Java) has to be used:

```
remote_agent.variable
```

where remote_agent can be the name of an agent, an expression returning an agent, self, myself or each. For instance, if we modify the previous species by giving its agents the possibility to feed another agent found in its neighbourhood, the result would be:

```
species animal {
   float energy <- 1000 min: 0 max: 2000 update: energy - 0.001;
   int age_in_years <- 1 update: age_in_years + int (time / 365);
   action eat (float amount <- 0.0) {
       float gain <- amount / age_in_years;
       energy <- energy + gain;
   }
   action feed (animal target){
       if (agent_to_feed != nil) and (agent_to_feed.energy <
   energy { // verifies that the agent exists and that it need to
    be fed
           ask agent_to_feed {
               do eat amount: myself.energy / 10; // asks the
   agent to eat 10% of our own energy
           }
           energy <- energy - (energy / 10); // reduces the
   energy by 10%
       }
   }
   reflex {
       animal candidates <- agents_overlapping (10 around agent.
   shape); gathers all the neighbours
```

```
        agent_to_feed value: candidates with_min_of (each.energy);
    //grabs one agent with the lowest energy
        do feed target: agent_to_feed; // tries to feed it
    }
}
```

In this example, agent_to_feed.energy, myself.energy and each.energy show different remote accesses to the attribute energy. The dotted notation used here can be employed in assignments as well. For instance, an action allowing two agents to exchange their energy could be defined as:

```
action random_exchange {//exchanges our energy with that of the
    closest agent
      animal one_agent <- agent_closest_to (self)/>
      float temp  <-one_agent.energy; // temporary storage of the
    agent's energy
      one_agent.energy <- energy; // assignment of the agent's
    energy with our energy
      energy <- temp;
}
```

Chapter 68

Operators

This file is automatically generated from java files. Do Not Edit It.

Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (`and`, `or`), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators: * if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`) * if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used. * if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any side effects on their operands. For instance, the `shuffle` operator, which randomizes the positions of elements in a list, does not modify its list operand but returns a new shuffled list.

Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely: * the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values. * it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else). * next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test) * next are the comparison operators (i.e. >, <, <=, >=, =, `!=`) * next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators) * next the unary operators `-` and `!` * next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points) * and finally the functional operators, which have the highest priority of all.

Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(
   my_agent, 10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

box, change_clockwise, cone3D, cube, cylinder, dem, hexagon, is_clockwise, pyramid, rgb_-to_xyz, set_z, sphere, teapot,

---

Arithmetic operators

-, /, [](#), *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,

---

BDI

and, eval_when, get_about, get_decay, get_intensity, get_lifetime, get_priority, get_super_intention, new_emotion, new_predicate, or, set_about, set_decay, set_intensity, set_-truth, with_lifetime, with_priority, with_values,

---

Casting operators

as, as_int, as_matrix, font, is, is_skill, list_with, matrix_with, species, to_gaml, topology,

---

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, grayscale, hsb, mean, median, rgb, rnd_-color, sum,

---

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, among, at, collect, contains, contains_all, contains_any, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_at, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

---

Date-related operators

-, +, add_days, add_hours, add_minutes, add_months, add_weeks, add_years, subtract_-days, subtract_hours, subtract_minutes, subtract_months, subtract_weeks, subtract_-years,

---

Driving operators

as_driving_graph,

---

edge

edge_between,

EDP-related operators

diff, diff2, internal_zero_order_equation,

Files-related operators

crs, file, file_exists, folder, get, new_folder, osm_file, read, writable,

---

FIPA-related operators

conversation, message,

Graphs-related operators

add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_-index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_watts_-strogatz, grid_cells_to_graph, in_degree_of, in_edges_of, layout, load_graph_from_file, load_shortest_paths, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_-from, rewire_n, source_of, spatial_graph, successors_of, sum, target_of, undirected, use_-cache, weight_of, with_optimizer_type, with_weights,

---

Grid-related operators

as_4_grid, as_grid, as_hexagonal_grid, grid_at, path_between,

---

Iterator operators

accumulate, as_map, collect, count, distribution_of, distribution_of, distribution_of, distribution2d_of, distribution2d_of, distribution2d_of, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, product_of, sort_by, sum_of, variance_of, where, with_max_of, with_min_of,

---

List-related operators

copy_between, index_of, last_index_of,

Logical operators

:, !, ?, and, or,

Map comparaison operators

fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,

---

Map-related operators

as_map, index_of, last_index_of,

Material

material,

Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

---

multicriteria operators

electre_DM, evidence_theory_DM, promethee_DM, weighted_means_DM,

---

Path-related operators

agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, path_between, path_to, paths_between, use_cache,

---

Points-related operators

-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, closest_points_with, farthest_point_to, grid_at, norm, point, points_at, points_on,

---

Random operators

binomial, flip, gauss, poisson, rnd, rnd_choice, sample, shuffle, truncated_gauss,

---

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

---

Spatial operators

-, *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, angle_between, any_location_in, arc, around, as_4_grid, as_grid, as_hexagonal_grid, at_distance, at_location, box, change_clockwise, circle, clean, closest_points_with, closest_to, cone, cone3D, convex_hull, covers, cross, crosses, crs, CRS_transform, cube, curve, cylinder, dem, direction_between, disjoint_from, distance_between, distance_to, ellipse, envelope, farthest_point_to, farthest_to, geometry_collection, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, is_clockwise, line, link, masked_by, neighbors_at, neighbors_of, overlapping, overlaps, partially_overlaps, path_between, path_to, plan, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rgb_to_xyz, rotated_by, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_squares, touches, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

---

Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, at_distance, closest_to, farthest_to, inside, neighbors_at, neighbors_of, overlapping,

---

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

---

Spatial statistical operators

hierarchical_clustering, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, convex_hull, CRS_transform, rotated_by, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_squares, transformed_by, translated_by, triangulate, voronoi, without_holes,

---

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

build, corR, dbscan, distribution_of, distribution2d_of, frequency_of, geometric_mean, harmonic_mean, hierarchical_clustering, kmeans, kurtosis, max, mean, mean_deviation, meanR, median, min, mul, predict, simple_clustering_by_distance, skewness, standard_-deviation, sum, variance,

---

Strings-related operators

+, <, <=, >, >=, as_date, as_system_date, as_system_time, as_time, at, char, contains, contains_all, contains_any, copy_between, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, replace, replace_regex, reverse, sample, shuffle, split_with, upper_case,

---

System

., command, copy, dead, eval_gaml, every, user_input,

Time-related operators

as_date, as_system_date, as_system_time, as_time,

Types-related operators

User control operators

user_input,

Operators

-

Possible use:

- - (`float`) —> `float`
- - (`int`) —> `int`
- `point` - `float` —> `point`
- - (`point` , `float`) —> `point`
- `map` - `map` —> `map`
- - (`map` , `map`) —> `map`
- `date` - `float` —> `date`
- - (`date` , `float`) —> `date`
- `container` - `container` —> `container`
- - (`container` , `container`) —> `container`
- `point` - `point` —> `point`
- - (`point` , `point`) —> `point`
- `point` - `int` —> `point`
- - (`point` , `int`) —> `point`
- `date` - `date` —> `float`
- - (`date` , `date`) —> `float`
- `date` - `int` —> `date`

- - (date , int) —> date
- map - pair —> map
- - (map , pair) —> map
- int - matrix —> matrix
- - (int , matrix) —> matrix
- container - unknown —> container
- - (container , unknown) —> container
- matrix - int —> matrix
- - (matrix , int) —> matrix
- rgb - rgb —> rgb
- - (rgb , rgb) —> rgb
- matrix - float —> matrix
- - (matrix , float) —> matrix
- matrix - matrix —> matrix
- - (matrix , matrix) —> matrix
- int - float —> float
- - (int , float) —> float
- geometry - container<geometry> —> geometry
- - (geometry , container<geometry>) —> geometry
- rgb - int —> rgb
- - (rgb , int) —> rgb
- geometry - geometry —> geometry
- - (geometry , geometry) —> geometry
- int - int —> int
- - (int , int) —> int
- species - agent —> container
- - (species , agent) —> container
- geometry - float —> geometry
- - (geometry , float) —> geometry
- float - int —> float
- - (float , int) —> float
- float - float —> float
- - (float , float) —> float
- float - matrix —> matrix
- - (float , matrix) —> matrix

Result:

If it is used as an unary operator, it returns the opposite of the operand. Returns the difference of the two operands.

Comment:

The behavior of the operator depends on the type of the operands.

Special cases:

- if both operands are containers and the right operand is empty, - returns the left operand
- if the left operand is a species and the right operand is an agent of the species, - returns a list containing all the agents of the species minus this agent
- if left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the difference of the operand coordinate with this number.

```
point var5 <- {1, 2} - 4.5;      // var5 equals {-3.5, -2.5, -4.5}
point var6 <- {1, 2} - 4;    // var6 equals {-3.0,-2.0,-4.0}
```

- if both operands are containers, returns a new list in which all the elements of the right operand have been removed from the left one

```
list<int> var7 <- [1,2,3,4,5,6] - [2,4,9];  // var7 equals
   [1,3,5,6]
list<int> var8 <- [1,2,3,4,5,6] - [0,8];    // var8 equals
   [1,2,3,4,5,6]
```

- if both operands are points, returns their difference (coordinates per coordinates).

```
point var9 <- {1, 2} - {4, 5};  // var9 equals {-3.0, -3.0}
```

- if both operands are dates, returns the duration in second between from date2 to date1

```
float var10 <- date1 - date2;   // var10 equals 598
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date minus the given number as duration (in seconds)

```
date1 - 200
```

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic difference of the number with each element of the matrix (results are float if the number is a float.

```
matrix var12 <- 3.5 - matrix([[2,5],[3,4]]);    // var12 equals
   matrix([[1.5,-1.5],[0.5,-0.5]])
```

- if the left operand is a list and the right operand is an object of any type (except list), - returns a list containing the elements of the left operand minus all the occurrences of this object

```
list<int> var13 <- [1,2,3,4,5,6] - 2;   // var13 equals
   [1,3,4,5,6]
list<int> var14 <- [1,2,3,4,5,6] - 0;   // var14 equals
   [1,2,3,4,5,6]
```

- if both operands are colors, returns a new color resulting from the subtraction of the two operands, component by component

```
rgb var15 <- rgb([255, 128, 32]) - rgb('red');  // var15 equals
   rgb([0,128,32])
```

- if the right-operand is a list of points, geometries or agents, returns the geometry resulting from the difference between the left-geometry and all of the right-geometries

```
geometry var16 <- rectangle(10,10) - [circle(2), square(2)];
   // var16 equals rectangle(10,10) - (circle(2) + square(2))
```

- if one operand is a color and the other an integer, returns a new color resulting from the subtraction of each component of the color with the right operand

```
rgb var17 <- rgb([255, 128, 32]) - 3;   // var17 equals rgb
   ([252,125,29])
```

- if both operands are a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries

```
geometry var18 <- geom1 - geom2;    // var18 equals a geometry
   corresponding to difference between geom1 and geom2
```

- if both operands are numbers, performs a normal arithmetic difference and returns a float if one of them is a float.

```
int var19 <- 1 - 1;     // var19 equals 0
int var20 <- 1.0 - 1;   // var20 equals 0.0
int var21 <- 3.7 - 1.2;     // var21 equals 2.5
int var22 <- 3 - 1.2;   // var22 equals 1.8
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) reduced by the right-hand operand distance

```
geometry var23 <- shape - 5;    // var23 equals a geometry
   corresponding to the geometry of the agent applying the
   operator reduced by a distance of 5
```

Examples:

```
map var0 <- ['a'::1,'b'::2] - ['b'::2];      // var0 equals ['a
    '::1]
map var1 <- ['a'::1,'b'::2] - ['b'::2,'c'::3];  // var1 equals ['
    a'::1]
map var2 <- ['a'::1,'b'::2] - ('b'::2);      // var2 equals ['a
    '::1]
map var3 <- ['a'::1,'b'::2] - ('c'::3);      // var3 equals ['a
    '::1,'b'::2]
int var4 <- - (-56);     // var4 equals 56
```

See also:

-, +, *, /,

:

Possible use:

- unknown : unknown —> unknown
- : (unknown , unknown) —> unknown

See also:

?,

::

Possible use:

- any expression :: any expression —> pair
- :: (any expression , any expression) —> pair

Result:

produces a new pair combining the left and the right operands

Special cases:

- nil is not acceptable as a key (although it is as a value). If such a case happens, :: will throw an appropriate error

---

!

Possible use:

- ! (`bool`) —> `bool`

Result:

opposite boolean value.

Special cases:

- if the parameter is not boolean, it is casted to a boolean value.

Examples:

```
bool var0 <- ! (true);  // var0 equals false
```

See also:

bool, and, or,

!=

Possible use:

- `unknown` != `unknown` —> `bool`
- != (`unknown` , `unknown`) —> `bool`
- `float` != `int` —> `bool`
- != (`float` , `int`) —> `bool`
- `int` != `float` —> `bool`
- != (`int` , `float`) —> `bool`
- `float` != `float` —> `bool`
- != (`float` , `float`) —> `bool`

Result:

true if both operands are different, false otherwise

Examples:

```
bool var0 <- [2,3] != [2,3];     // var0 equals false
bool var1 <- [2,4] != [2,3];     // var1 equals true
bool var2 <- 3.0 != 3;  // var2 equals false
bool var3 <- 4.7 != 4;  // var3 equals true
bool var4 <- 3 != 3.0;  // var4 equals false
bool var5 <- 4 != 4.7;  // var5 equals true
bool var6 <- 3.0 != 3.0;     // var6 equals false
bool var7 <- 4.0 != 4.7;     // var7 equals true
```

See also:

=, >, <, >=, <=,

?

Possible use:

- `bool` ? `any expression` —> `unknown`
- ? (`bool` , `any expression`) —> `unknown`

Result:

It is used in combination with the : operator: if the left-hand operand evaluates to true, returns the value of the left-hand operand of the :, otherwise that of the right-hand operand of the :

Comment:

These functional tests can be combined together.

Examples:

```
list<string> var0 <- [10, 19, 43, 12, 7, 22] collect ((each > 20)
    ? 'above' : 'below');     // var0 equals ['below', 'below', '
   above', 'below', 'below', 'above']
rgb color <- (flip(0.3) ? #red : (flip(0.9) ? #blue : #green));
```

See also:

:,

/

Possible use:

- `matrix` / `matrix` —> `matrix`

- / (`matrix` , `matrix`) —> `matrix`
- `int` / `int` —> `float`
- / (`int` , `int`) —> `float`
- `rgb` / `int` —> `rgb`
- / (`rgb` , `int`) —> `rgb`
- `point` / `float` —> `point`
- / (`point` , `float`) —> `point`
- `float` / `float` —> `float`
- / (`float` , `float`) —> `float`
- `point` / `int` —> `point`
- / (`point` , `int`) —> `point`
- `matrix` / `float` —> `matrix`
- / (`matrix` , `float`) —> `matrix`
- `matrix` / `int` —> `matrix`
- / (`matrix` , `int`) —> `matrix`
- `float` / `int` —> `float`
- / (`float` , `int`) —> `float`
- `rgb` / `float` —> `rgb`
- / (`rgb` , `float`) —> `rgb`
- `int` / `float` —> `float`
- / (`int` , `float`) —> `float`

Result:

Returns the division of the two operands.

Special cases:

- if the right-hand operand is equal to zero, raises a "Division by zero" exception

- if both operands are numbers (float or int), performs a normal arithmetic division and returns a float.

```
float var0 <- 3 / 5.0;  // var0 equals 0.6
```

- if one operand is a color and the other an integer, returns a new color resulting from the division of each component of the color by the right operand

```
rgb var1 <- rgb([255, 128, 32]) / 2;    // var1 equals rgb
   ([127,64,16])
```

- if the left operand is a point, returns a new point with coordinates divided by the right operand

```
point var2 <- {5, 7.5} / 2.5;   // var2 equals {2, 3}
point var3 <- {2,5} / 4;    // var3 equals {0.5,1.25}
```

- if one operand is a color and the other a double, returns a new color resulting from the division of each component of the color by the right operand. The result on each component is then truncated.

```
rgb var4 <- rgb([255, 128, 32]) / 2.5;  // var4 equals rgb
   ([102,51,13])
```

See also:

+, -, *,

.

Possible use:

- `agent` . `any expression` —> `unknown`
- . (`agent` , `any expression`) —> `unknown`
- `matrix` . `matrix` —> `matrix`
- . (`matrix` , `matrix`) —> `matrix`

Result:

It has two different uses: it can be the dot product between 2 matrices or return an evaluation of the expression (right-hand operand) in the scope the given agent.

Special cases:

- if the agent is nil or dead, throws an exception
- if the left operand is an agent, it evaluates of the expression (right-hand operand) in the scope the given agent

```
unknown var0 <- agent1.location;    // var0 equals the location
   of the agent agent1
map(nil).keys
```

- if both operands are matrix, returns the dot product of them

```
matrix var2 <- matrix([[1,1],[1,2]]) . matrix([[1,1],[1,2]]);
   // var2 equals matrix([[2,3],[3,5]])
```

---

^

Possible use:

- `int` ^ `int` —> `float`
- ^ (`int` , `int`) —> `float`
- `int` ^ `float` —> `float`
- ^ (`int` , `float`) —> `float`
- `float` ^ `float` —> `float`
- ^ (`float` , `float`) —> `float`
- `float` ^ `int` —> `float`
- ^ (`float` , `int`) —> `float`

Result:

Returns the value (always a float) of the left operand raised to the power of the right operand.

Special cases:

- if the right-hand operand is equal to 0, returns 1
- if it is equal to 1, returns the left-hand operand.
- Various examples of power

```
float var0 <- 2 ^ 3;     // var0 equals 8.0
```

Examples:

```
float var12 <- 4.84 ^ 0.5;  // var12 equals 2.2
```

See also:

*, sqrt,

@

Same signification as at

---

*

Possible use:

- `matrix` * `float` —> `matrix`
- * (`matrix` , `float`) —> `matrix`
- `point` * `float` —> `point`
- * (`point` , `float`) —> `point`

- `float` * `float` —> `float`
- * (`float` , `float`) —> `float`
- `int` * `matrix` —> `matrix`
- * (`int` , `matrix`) —> `matrix`
- `float` * `matrix` —> `matrix`
- * (`float` , `matrix`) —> `matrix`
- `int` * `int` —> `int`
- * (`int` , `int`) —> `int`
- `geometry` * `float` —> `geometry`
- * (`geometry` , `float`) —> `geometry`
- `int` * `float` —> `float`
- * (`int` , `float`) —> `float`
- `matrix` * `matrix` —> `matrix`
- * (`matrix` , `matrix`) —> `matrix`
- `matrix` * `int` —> `matrix`
- * (`matrix` , `int`) —> `matrix`
- `geometry` * `point` —> `geometry`
- * (`geometry` , `point`) —> `geometry`
- `rgb` * `int` —> `rgb`
- * (`rgb` , `int`) —> `rgb`
- `point` * `int` —> `point`
- * (`point` , `int`) —> `point`
- `point` * `point` —> `float`
- * (`point` , `point`) —> `float`
- `float` * `int` —> `float`
- * (`float` , `int`) —> `float`

Result:

Returns the product of the two operands.

Special cases:

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic product of the number with each element of the matrix (results are float if the number is a float.

```
matrix<float> m <- (3.5 * matrix([[2,5],[3,4]]));   //m equals
   matrix([[7.0,17.5],[10.5,14]])
```

- if both operands are numbers (float or int), performs a normal arithmetic product and returns a float if one of them is a float.

```
int var1 <- 1 * 1;  // var1 equals 1
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) scaled by the right-hand operand coefficient

```
geometry var2 <- circle(10) * 2;    // var2 equals circle(20)
```

- if the left-hand operand is a geometry and the right-hand operand a point, returns a geometry corresponding to the left-hand operand (geometry, agent, point) scaled by the right-hand operand coefficients in the 3 dimensions

```
geometry var3 <- shape * {0.5,0.5,2};   // var3 equals a geometry
    corresponding to the geometry of the agent applying the
   operator scaled by a coefficient of 0.5 in x, 0.5 in y and 2
   in z
```

- if one operand is a color and the other an integer, returns a new color resulting from the product of each component of the color with the right operand (with a maximum value at 255)

```
rgb var4 <- rgb([255, 128, 32]) * 2;    // var4 equals rgb
   ([255,255,64])
```

- if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number

```
point var5 <- {2,5} * 4;     // var5 equals {8.0, 20.0}
point var6 <- {2, 4} * 2.5;      // var6 equals {5.0, 10.0}
```

- if both operands are points, returns their scalar product

```
float var7 <- {2,5} * {4.5, 5};      // var7 equals 34.0
```

Examples:

```
float var8 <- 2.5 * 2;  // var8 equals 5.0
```

See also:

/, +, -,

+

Possible use:

- rgb + rgb —> rgb
- + (rgb , rgb) —> rgb
- geometry + geometry —> geometry
- + (geometry , geometry) —> geometry
- container + unknown —> container
- + (container , unknown) —> container
- float + int —> float
- + (float , int) —> float
- string + string —> string
- + (string , string) —> string
- int + matrix —> matrix
- + (int , matrix) —> matrix

- date + float —> date
- + (date , float) —> date
- rgb + int —> rgb
- + (rgb , int) —> rgb
- point + int —> point
- + (point , int) —> point
- float + date —> date
- + (float , date) —> date
- map + map —> map
- + (map , map) —> map
- map + pair —> map
- + (map , pair) —> map
- geometry + float —> geometry
- + (geometry , float) —> geometry
- float + matrix —> matrix
- + (float , matrix) —> matrix
- date + int —> date
- + (date , int) —> date
- matrix + matrix —> matrix
- + (matrix , matrix) —> matrix
- container + container —> container
- + (container , container) —> container
- string + unknown —> string
- + (string , unknown) —> string
- point + float —> point
- + (point , float) —> point
- int + int —> int
- + (int , int) —> int
- point + point —> point
- + (point , point) —> point
- matrix + float —> matrix
- + (matrix , float) —> matrix
- int + date —> date
- + (int , date) —> date
- int + float —> float
- + (int , float) —> float
- date + string —> string
- + (date , string) —> string

- `matrix` + `int` —> `matrix`
- + (`matrix` , `int`) —> `matrix`
- `float` + `float` —> `float`
- + (`float` , `float`) —> `float`
- + (`geometry`, `float`, `int`) —> `geometry`
- + (`geometry`, `float`, `int`, `int`) —> `geometry`

Result:

Returns the sum, union or concatenation of the two operands.

Special cases:

- if one of the operands is nil, + throws an error

- if both operands are species, returns a special type of list called meta-population

- if both operands are colors, returns a new color resulting from the sum of the two operands, component by component

```
rgb var4 <- rgb([255, 128, 32]) + rgb('red');   // var4 equals
   rgb([255,128,32])
```

- if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the union between both geometries

```
geometry var5 <- geom1 + geom2;     // var5 equals a geometry
   corresponding to union between geom1 and geom2
```

- if the right operand is an object of any type (except a container), + returns a list of the elements of the left operand, to which this object has been added

```
list<int> var6 <- [1,2,3,4,5,6] + 2;    // var6 equals
   [1,2,3,4,5,6,2]
list<int> var7 <- [1,2,3,4,5,6] + 0;    // var7 equals
   [1,2,3,4,5,6,0]
```

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic sum of the number with each element of the matrix (results are float if the number is a float.

```
matrix var8 <- 3.5 + matrix([[2,5],[3,4]]);     // var8 equals
   matrix([[5.5,8.5],[6.5,7.5]])
```

- if one operand is a color and the other an integer, returns a new color resulting from the sum of each component of the color with the right operand

```
rgb var9 <- rgb([255, 128, 32]) + 3;    // var9 equals rgb
   ([255,131,35])
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the right-hand operand distance. The number of segments used by default is 8 and the end cap style is #round

```
geometry var10 <- circle(5) + 5;    // var10 equals circle(10)
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date plus the given number as duration (in seconds)

```
date1 + 200
```

- if both operands are list, +returns the concatenation of both lists.

```
list<int> var12 <- [1,2,3,4,5,6] + [2,4,9];     // var12 equals
   [1,2,3,4,5,6,2,4,9]
list<int> var13 <- [1,2,3,4,5,6] + [0,8];   // var13 equals
   [1,2,3,4,5,6,0,8]
```

- if the left-hand operand is a geometry and the right-hand operands a float and an integer, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand

```
geometry var14 <- circle(5) + (5,32);   // var14 equals circle
   (10)
```

- if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one beind casted into a string)

```
string var15 <- "hello " + 12;  // var15 equals "hello 12"
```

- if the left-hand operand is a geometry and the right-hand operands a float, an integer and one of #round, #square or #flat, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand and a flat, square or round end cap style

```
geometry var16 <- circle(5) + (5,32,#round);    // var16 equals
   circle(10)
```

- if the left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the sum of the operand coordinate with this number.

```
point var17 <- {1, 2} + 4;  // var17 equals {5.0, 6.0,4.0}
point var18 <- {1, 2} + 4.5;    // var18 equals {5.5, 6.5,4.5}
```

- if both operands are numbers (float or int), performs a normal arithmetic sum and returns a float if one of them is a float.

```
int var19 <- 1 + 1;     // var19 equals 2
int var20 <- 1.0 + 1;   // var20 equals 2.0
int var21 <- 1.0 + 2.5;     // var21 equals 3.5
```

- if both operands are points, returns their sum.

```
point var22 <- {1, 2} + {4, 5};     // var22 equals {5.0, 7.0}
```

Examples:

```
map var0 <- ['a'::1,'b'::2] + ['c'::3];      // var0 equals ['a
   '::1,'b'::2,'c'::3]
map var1 <- ['a'::1,'b'::2] + [5::3.0];      // var1 equals ['a
   '::1.0,'b'::2.0,5::3.0]
map var2 <- ['a'::1,'b'::2] + ('c'::3);      // var2 equals ['a
   '::1,'b'::2,'c'::3]
map var3 <- ['a'::1,'b'::2] + ('c'::3);      // var3 equals ['a
   '::1,'b'::2,'c'::3]
```

See also:

-, /, *,

<

Possible use:

- `string` < `string` —> `bool`
- < (`string` , `string`) —> `bool`
- `float` < `float` —> `bool`
- < (`float` , `float`) —> `bool`
- `int` < `float` —> `bool`
- < (`int` , `float`) —> `bool`
- `int` < `int` —> `bool`
- < (`int` , `int`) —> `bool`
- `float` < `int` —> `bool`
- < (`float` , `int`) —> `bool`
- `point` < `point` —> `bool`
- < (`point` , `point`) —> `bool`

Result:

true if the left-hand operand is less than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var4 <- 'abc' < 'aeb';      // var4 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var5 <- {5,7} < {4,6};      // var5 equals false
bool var6 <- {5,7} < {4,8};      // var6 equals false
```

Examples:

```
bool var0 <- 3.5 < 7.6;      // var0 equals true
bool var1 <- 3 < 2.5;    // var1 equals false
bool var2 <- 3 < 7;      // var2 equals true
bool var3 <- 3.5 < 7;    // var3 equals true
```

See also:

>, >=, <=, =, !=,

<=

Possible use:

- `float` <= `float` —> `bool`
- <= (`float` , `float`) —> `bool`

- int <= int —> bool
- <= (int , int) —> bool
- string <= string —> bool
- <= (string , string) —> bool
- float <= int —> bool
- <= (float , int) —> bool
- point <= point —> bool
- <= (point , point) —> bool
- int <= float —> bool
- <= (int , float) —> bool

Result:

true if the left-hand operand is less or equal than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' <= 'aeb';    // var0 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var1 <- {5,7} <= {4,6};    // var1 equals false
bool var2 <- {5,7} <= {4,8};    // var2 equals false
```

Examples:

```
bool var3 <- 3.5 <= 3.5;    // var3 equals true
bool var4 <- 3 <= 7;    // var4 equals true
bool var5 <- 7.0 <= 7;  // var5 equals true
bool var6 <- 3 <= 2.5;  // var6 equals false
```

See also:

>, <, >=, =, !=,

<>

Same signification as !=

=

Possible use:

- `float` = `float` —> `bool`
- = (`float` , `float`) —> `bool`
- `unknown` = `unknown` —> `bool`
- = (`unknown` , `unknown`) —> `bool`
- `int` = `float` —> `bool`
- = (`int` , `float`) —> `bool`
- `int` = `int` —> `bool`
- = (`int` , `int`) —> `bool`
- `float` = `int` —> `bool`
- = (`float` , `int`) —> `bool`

Result:

returns true if both operands are equal, false otherwise returns true if both operands are equal, false otherwise

Special cases:

- if both operands are any kind of objects, returns true if they are identical (i.e., the same object) or equal (comparisons between nil values are permitted)

```
bool var0 <- [2,3] = [2,3];      // var0 equals true
```

Examples:

```
bool var1 <- 4.5 = 4.7;      // var1 equals false
bool var2 <- 3 = 3.0;    // var2 equals true
bool var3 <- 4 = 4.7;    // var3 equals false
bool var4 <- 4 = 5;      // var4 equals false
bool var5 <- 4.7 = 4;    // var5 equals false
```

See also:

>, <, >=, <=, !=,

>

Possible use:

- `point` > `point` —> `bool`
- > (`point` , `point`) —> `bool`
- `float` > `int` —> `bool`
- > (`float` , `int`) —> `bool`
- `int` > `float` —> `bool`
- > (`int` , `float`) —> `bool`
- `string` > `string` —> `bool`
- > (`string` , `string`) —> `bool`
- `float` > `float` —> `bool`
- > (`float` , `float`) —> `bool`
- `int` > `int` —> `bool`
- > (`int` , `int`) —> `bool`

Result:

true if the left-hand operand is greater than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are points, returns true if and only if the left component (x) of the left operand if greater than x of the right one and if the right component (y) of the left operand is greater than y of the right one.

```
bool var0 <- {5,7} > {4,6};     // var0 equals true
bool var1 <- {5,7} > {4,8};     // var1 equals false
```

- if both operands are String, uses a lexicographic comparison of two strings

```
bool var2 <- 'abc' > 'aeb';     // var2 equals false
```

Examples:

```
bool var3 <- 3.5 > 7;    // var3 equals false
bool var4 <- 3 > 2.5;    // var4 equals true
bool var5 <- 3.5 > 7.6;     // var5 equals false
bool var6 <- 3 > 7;      // var6 equals false
```

See also:

<, >=, <=, =, !=,

>=

Possible use:

- `int` >= `float` —> `bool`
- >= (`int` , `float`) —> `bool`
- `int` >= `int` —> `bool`
- >= (`int` , `int`) —> `bool`
- `float` >= `float` —> `bool`
- >= (`float` , `float`) —> `bool`
- `point` >= `point` —> `bool`
- >= (`point` , `point`) —> `bool`
- `float` >= `int` —> `bool`
- >= (`float` , `int`) —> `bool`
- `string` >= `string` —> `bool`
- >= (`string` , `string`) —> `bool`

Result:

true if the left-hand operand is greater or equal than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are points, returns true if and only if the left component (x) of the left operand if greater or equal than x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var4 <- {5,7} >= {4,6};     // var4 equals true
bool var5 <- {5,7} >= {4,8};     // var5 equals false
```

- if both operands are string, uses a lexicographic comparison of the two strings

```
bool var6 <- 'abc' >= 'aeb';     // var6 equals false
bool var7 <- 'abc' >= 'abc';     // var7 equals true
```

Examples:

```
bool var0 <- 3 >= 2.5;  // var0 equals true
bool var1 <- 3 >= 7;    // var1 equals false
bool var2 <- 3.5 >= 3.5;    // var2 equals true
bool var3 <- 3.5 >= 7;  // var3 equals false
```

See also:

>, <, <=, =, !=,

---

abs

Possible use:

- abs (float) —> float
- abs (int) —> int

Result:

Returns the absolute value of the operand (so a positive int or float depending on the type of the operand).

Examples:

```
float var0 <- abs (200 * -1 + 0.5);     // var0 equals 199.5
int var1 <- abs (-10);  // var1 equals 10
int var2 <- abs (10);   // var2 equals 10
```

---

accumulate

Possible use:

- container accumulate any expression —> container
- accumulate (container , any expression) —> container

Result:

returns a new flat list, in which each element is the evaluation of the right-hand operand. If this evaluation returns a list, the elements of this result are added directly to the list returned

Comment:

accumulate is dedicated to the application of a same computation on each element of a container (and returns a list). In the right-hand operand, the keyword each can be used to represent, in turn, each of the left-hand operand elements.

Examples:

```
container var0 <- [a1,a2,a3] accumulate (each neighbors_at 10);
      // var0 equals a flat list of all the neighbors of these
   three agents
list<int> var1 <- [1,2,4] accumulate ([2,4]);   // var1 equals
   [2,4,2,4,2,4]
list<int> var2 <- [1,2,4] accumulate (each * 2);    // var2
   equals [2,4,8]
```

See also:

collect,

`acos`

Possible use:

- `acos (float) —> float`
- `acos (int) —> float`

Result:

Returns the value (in the interval [0,180], in decimal degrees) of the arccos of the operand (which should be in [-1,1]).

Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

Examples:

```
float var0 <- acos (0);      // var0 equals 90.0
```

See also:

asin, atan, cos,

`add_days`

Possible use:

- `date add_days int —> date`
- `add_days (date , int) —> date`

Result:

Add a given number of days to a date

Examples:

```
date1 add_days 20
```

---

add_edge

Possible use:

- graph add_edge pair —> graph
- add_edge (graph , pair) —> graph

Result:

add an edge between a source vertex and a target vertex (resp. the left and the right element of the pair operand)

Comment:

if the edge already exists, the graph is unchanged

Examples:

```
graph <- graph add_edge (source::target);
```

See also:

add_node, graph,

---

add_hours

Possible use:

- date add_hours int —> date
- add_hours (date , int) —> date

Result:

Add a given number of hours to a date

Examples:

```
date1 add_hours 15
```

---

add_minutes

Possible use:

- date add_minutes int —> date
- add_minutes (date , int) —> date

Result:

Add a given number of minutes to a date

Examples:

```
date1 add_minutes 5
```

---

add_months

Possible use:

- date add_months int —> date
- add_months (date , int) —> date

Result:

Add a given number of months to a date

Examples:

```
date1 add_months 5
```

---

add_node

Possible use:

- graph add_node geometry —> graph
- add_node (graph , geometry) —> graph

Result:

adds a node in a graph.

Examples:

```
graph var0 <- graph add_node node(0) ;  // var0 equals the graph
    with node(0)
```

See also:

add_edge, graph,

---

add_point

Possible use:

- `geometry add_point point —> geometry`
- `add_point (geometry , point) —> geometry`

Result:

A new geometry resulting from the addition of the right point (coordinate) to the left-hand geometry. Note that adding a point to a line or polyline will always return a closed contour. Also note that the position at which the added point will appear in the geometry is not necessarily the last one, as points are always ordered in a clockwise fashion in geometries

Examples:

```
geometry var0 <- polygon([{10,10},{10,20},{20,20}]) add_point
   {20,10};  // var0 equals polygon
   ([{10,10},{10,20},{20,20},{20,10}])
```

---

add_seconds

Same signification as +

`add_weeks`

Possible use:

- `date` `add_weeks` `int` —> `date`
- `add_weeks` (`date` , `int`) —> `date`

Result:

Add a given number of weeks to a date

Examples:

```
date1 add_weeks 15
```

`add_years`

Possible use:

- `date` `add_years` `int` —> `date`
- `add_years` (`date` , `int`) —> `date`

Result:

Add a given number of year to a date

Examples:

```
date1 add_years 3
```

`adjacency`

Possible use:

- `adjacency` (`graph`) —> `matrix<float>`

Result:

adjacency matrix of the given graph.

`agent`

Possible use:

- `agent` (`any`) —> `agent`

Result:

Casts the operand into the type agent

`agent_closest_to`

Possible use:

- `agent_closest_to` (`unknown`) —> `agent`

Result:

An agent, the closest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
agent var0 <- agent_closest_to(self);    // var0 equals the
   closest agent to the agent applying the operator.
```

See also:

neighbors_at, neighbors_of, agents_inside, agents_overlapping, closest_to, inside, overlapping,

---

`agent_farthest_to`

Possible use:

- `agent_farthest_to` (`unknown`) —> `agent`

Result:

An agent, the farthest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
agent var0 <- agent_farthest_to(self);   // var0 equals the
   farthest agent to the agent applying the operator.
```

See also:

neighbors_at, neighbors_of, agents_inside, agents_overlapping, closest_to, inside, overlapping, agent_closest_to, farthest_to,

---

agent_from_geometry

Possible use:

- path agent_from_geometry geometry —> agent
- agent_from_geometry (path , geometry) —> agent

Result:

returns the agent corresponding to given geometry (right-hand operand) in the given path (left-hand operand).

Special cases:

- if the left-hand operand is nil, returns nil

Examples:

```
geometry line <- one_of(path_followed.segments);
road ag <- road(path_followed agent_from_geometry line);
```

See also:

path,

---

agents_at_distance

Possible use:

- agents_at_distance (float) —> container

Result:

A list of agents situated at a distance lower than the right argument.

Examples:

```
container var0 <- agents_at_distance(20);   // var0 equals all
   the agents (excluding the caller) which distance to the caller
    is lower than 20
```

See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping, at_distance,

---

agents_inside

Possible use:

- agents_inside (unknown) —> list<agent>

Result:

A list of agents covered by the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_inside(self);    // var0 equals the
   agents that are covered by the shape of the agent applying the
    operator.
```

See also:

agent_closest_to, agents_overlapping, closest_to, inside, overlapping,

---

agents_overlapping

Possible use:

- `agents_overlapping` (`unknown`) —> `list<agent>`

Result:

A list of agents overlapping the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_overlapping(self);   // var0 equals
   the agents that overlap the shape of the agent applying the
   operator.
```

See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping, at_distance,

---

all_pairs_shortest_path

Possible use:

- all_pairs_shortest_path (graph) —> matrix<int>

Result:

returns the successor matrix of shortest paths between all node pairs (rows: source, columns: target): a cell (i,j) will thus contains the next node in the shortest path between i and j.

Examples:

```
matrix<int> var0 <- all_pairs_shortest_paths(my_graph);     //
   var0 equals shortest_paths_matrix will contain all pairs of
   shortest paths
```

---

alpha_index

Possible use:

- alpha_index (graph) —> float

Result:

returns the alpha index of the graph (measure of connectivity which evaluates the number of cycles in a graph in comparison with the maximum number of cycles. The higher the alpha index, the more a network is connected: alpha = nb_cycles / (2*S-5) - planar graph)

Examples:

```
float var1 <- alpha_index(graphEpidemio);    // var1 equals the
   alpha index of the graph
```

See also:

beta_index, gamma_index, nb_cycles, connectivity_index,

---

among

Possible use:

- `int among container —> container`
- `among (int , container) —> container`

Result:

Returns a list of length the value of the left-hand operand, containing random elements from the right-hand operand. As of GAMA 1.6, the order in which the elements are returned can be different than the order in which they appear in the right-hand container

Special cases:

- if the right-hand operand is empty, among returns a new empty list. If it is nil, it throws an error.

- if the left-hand operand is greater than the length of the right-hand operand, among returns the right-hand operand (converted as a list). If it is smaller or equal to zero, it returns an empty list

Examples:

```
list<int> var0 <- 3 among [1,2,4,3,5,7,6,8];     // var0 equals
   [1,2,8] (for example)
container var1 <- 3 among g2;    // var1 equals [node6,node11,
   node7]
container var2 <- 3 among list(node);    // var2 equals [node1,
   node11,node4]
list<int> var3 <- 1 among [1::2,3::4];   // var3 equals 2 or 4
```

---

and

Possible use:

- `bool and any expression —> bool`
- `and (bool , any expression) —> bool`

Result:

a bool value, equal to the logical and between the left-hand operand and the right-hand operand.

Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like (1 and 0) is accepted and returns false.

See also:

bool, or, !,

---

and

Possible use:

- `predicate` `and` `predicate` —> `predicate`
- `and` (`predicate` , `predicate`) —> `predicate`

Result:

create a new predicate from two others by including them as subintentions

Examples:

```
predicate1 and predicate2
```

---

angle_between

Possible use:

- `angle_between` (`point`, `point`, `point`) —> `int`

Result:

the angle between vectors P0P1 and P0P2 (P0, P1, P2 being the three point operands)

Examples:

```
int var0 <- angle_between({5,5},{10,5},{5,10});     // var0
   equals 90
```

---

any

Same signification as one_of

---

any_location_in

Possible use:

- any_location_in (geometry) —> point

Result:

A point inside (or touching) the operand-geometry.

Examples:

```
point var0 <- any_location_in(square(5));   // var0 equals a
   point in the square, for example : {3,4.6}.
```

See also:

closest_points_with, farthest_point_to, points_at,

---

any_point_in

Same signification as any_location_in

---

append_horizontally

Possible use:

- matrix append_horizontally matrix —> matrix
- append_horizontally (matrix , matrix) —> matrix
- matrix append_horizontally matrix —> matrix
- append_horizontally (matrix , matrix) —> matrix

Result:

A matrix resulting from the concatenation of the rows of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

Examples:

```
matrix var0 <- matrix([[1.0,2.0],[3.0,4.0]]) append_horizontally
   matrix([[1,2],[3,4]]);      // var0 equals matrix
   ([[1.0,2.0],[3.0,4.0],[1.0,2.0],[3.0,4.0]])
```

---

append_vertically

Possible use:

- matrix append_vertically matrix —> matrix
- append_vertically (matrix , matrix) —> matrix
- matrix append_vertically matrix —> matrix
- append_vertically (matrix , matrix) —> matrix

Result:

A matrix resulting from the concatenation of the columns of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

Examples:

```
matrix var0 <- matrix([[1,2],[3,4]]) append_vertically matrix
   ([[1,2],[3,4]]);   // var0 equals matrix
   ([[1,2,1,2],[3,4,3,4]])
```

`arc`

Possible use:

- `arc` (`float`, `float`, `float`) —> `geometry`
- `arc` (`float`, `float`, `float`, `bool`) —> `geometry`

Result:

An arc, which radius is equal to the first operand, heading to the second, amplitude to the third and a boolean indicating whether to return a linestring or a polygon to the fourth An arc, which radius is equal to the first operand, heading to the second and amplitude the third

Comment:

the center of the arc is by default the location of the current agent in which has been called this operator.the center of the arc is by default the location of the current agent in which has been called this operator. This operator returns a polygon by default.

Special cases:

- returns a point if the radius operand is lower or equal to 0.
- returns a point if the radius operand is lower or equal to 0.

Examples:

```
geometry var0 <- arc(4,45,90, false);   // var0 equals a geometry
    as an arc of radius 4, in a direction of 45Ã Â° and an
   amplitude of 90Ã Â°, which only contains the points on the arc
geometry var1 <- arc(4,45,90);  // var1 equals a geometry as an
   arc of radius 4, in a direction of 45Ã Â° and an amplitude of
   90Ã Â°
```

See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

around

Possible use:

- `float` `around` `unknown` —> `geometry`
- `around` (`float` , `unknown`) —> `geometry`

Result:

A geometry resulting from the difference between a buffer around the right-operand casted in geometry at a distance left-operand (right-operand buffer left-operand) and the right-operand casted as geometry.

Special cases:

- returns a circle geometry of radius right-operand if the left-operand is nil

Examples:

```
geometry var0 <- 10 around circle(5);   // var0 equals the ring
   geometry between 5 and 10.
```

See also:

circle, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

as

Possible use:

- `unknown` as `any expression` —> `unknown`
- as (`unknown` , `any expression`) —> `unknown`

Result:

casting of the first argument into a given type

Comment:

It is equivalent to the application of the type operator on the left operand.

Examples:

```
int var0 <- 3.5 as int;      // var0 equals int(3.5)
```

---

as_4_grid

Possible use:

- geometry as_4_grid point —> matrix
- as_4_grid (geometry , point) —> matrix

Result:

A matrix of square geometries (grid with 4-neighborhood) with dimension given by the right-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

Examples:

```
matrix var0 <- self as_4_grid {10, 5};  // var0 equals the matrix
    of square geometries (grid with 4-neighborhood) with 10
   columns and 5 lines corresponding to the square tessellation
   of the geometry of the agent applying the operator.
```

See also:

as_grid, as_hexagonal_grid,

as_date

Possible use:

- as_date (float) —> string
- float as_date string —> string
- as_date (float , string) —> string

Result:

converts a number of seconds in the model (for instance, the 'time' variable) into a string that represents the period elapsed since the beginning of the simulation using year, month, day, hour, minutes and seconds following a given pattern (right-hand operand). GAMA uses a special calendar for internal model times, where months have 30 days and years 12 months.

Special cases:

- used as an unary operator, it uses a defined pattern with years, months, days

```
string var0 <- as_date(22324234);   // var0 equals "8 months, 18
   days"
```

- Pattern should include : "%Y %M %D %h %m %s" for outputting years, months, days, hours, minutes, seconds

```
string var1 <- 22324234 as_date "%M m %D d %h h %m m %s seconds";
      // var1 equals "8 m 18 d 9 h 10 m 34 seconds"
```

See also:

as_time,

---

as_distance_graph

Possible use:

- `container` as_distance_graph `map` —> `graph`
- as_distance_graph (`container` , `map`) —> `graph`
- `container` as_distance_graph `float` —> `graph`
- as_distance_graph (`container` , `float`) —> `graph`
- as_distance_graph (`container`, `float`, `species`) —> `graph`

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices close enough (less than a distance, right-hand operand).

Comment:

as_distance_graph is more efficient for a list of points than as_intersection_graph.

Examples:

```
list(ant) as_distance_graph 3.0
```

See also:

as_intersection_graph, as_edge_graph,

`as_driving_graph`

Possible use:

- `container` `as_driving_graph` `container` —> `graph`
- `as_driving_graph` (`container` , `container`) —> `graph`

Result:

creates a graph from the list/map of edges given as operand and connect the node to the edge

Examples:

```
as_driving_graph(road,node)  --:  build a graph while using the
   road agents as edges and the node agents as nodes
```

See also:

as_intersection_graph, as_distance_graph, as_edge_graph,

---

as_edge_graph

Possible use:

- `as_edge_graph` (`map`) —> `graph`
- `as_edge_graph` (`container`) —> `graph`
- `container` `as_edge_graph` `float` —> `graph`
- `as_edge_graph` (`container` , `float`) —> `graph`

Result:

creates a graph from the list/map of edges given as operand

Special cases:

- if the operand is a map, the graph will be built by creating edges from pairs of the map

```
graph var0 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
      // var0 equals a graph with these three vertices and two
   edges
```

- if the operand is a list and a tolerance (max distance in meters to consider that 2 points are the same node) is given, the graph will be built with elements of the list as edges and two edges will be connected by a node if the distance between their extremity (first or last points) are at distance lower or equal to the tolerance

```
graph var1 <- as_edge_graph([line([{1,5},{12,45}]),line
   ([{13,45},{34,56}])],1);;    // var1 equals a graph with two
   edges and three vertices
```

- if the operand is a list, the graph will be built with elements of the list as edges

```
graph var2 <- as_edge_graph([line([{1,5},{12,45}]),line
   ([{12,45},{34,56}])]);   // var2 equals a graph with two edges
    and three vertices
```

See also:

as_intersection_graph, as_distance_graph,

---

as_grid

Possible use:

- `geometry` `as_grid` `point` —> `matrix`
- `as_grid` (`geometry` , `point`) —> `matrix`

Result:

A matrix of square geometries (grid with 8-neighborhood) with dimension given by the right-hand operand ({nb_cols, nb_lines}) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

Examples:

```
matrix var0 <- self as_grid {10, 5};    // var0 equals a matrix
   of square geometries (grid with 8-neighborhood) with 10
   columns and 5 lines corresponding to the square tessellation
   of the geometry of the agent applying the operator.
```

See also:

as_4_grid, as_hexagonal_grid,

---

as_hexagonal_grid

Possible use:

- `geometry` **`as_hexagonal_grid`** `point` —> `list<geometry>`
- **`as_hexagonal_grid`** (`geometry` , `point`) —> `list<geometry>`

Result:

A list of geometries (hexagonal) corresponding to the hexagonal tesselation of the first operand geometry

Examples:

```
list<geometry> var0 <- self as_hexagonal_grid {10, 5};  // var0
   equals list of geometries (hexagonal) corresponding to the
   hexagonal tesselation of the first operand geometry
```

See also:

as_4_grid, as_grid,

---

as_int

Possible use:

- `string` **`as_int`** `int` —> `int`
- **`as_int`** (`string` , `int`) —> `int`

Result:

parses the string argument as a signed integer in the radix specified by the second argument.

Special cases:

- if the left operand is nil or empty, as_int returns 0
- if the left operand does not represent an integer in the specified radix, as_int throws an exception

Examples:

```
int var0 <- '20' as_int 10;     // var0 equals 20
int var1 <- '20' as_int 8;  // var1 equals 16
int var2 <- '20' as_int 16;     // var2 equals 32
int var3 <- '1F' as_int 16;     // var3 equals 31
int var4 <- 'hello' as_int 32;  // var4 equals 18306744
```

See also:

int,

---

as_intersection_graph

Possible use:

- container as_intersection_graph float —> graph
- as_intersection_graph (container , float) —> graph

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices with an intersection (with a given tolerance).

Comment:

as_intersection_graph is more efficient for a list of geometries (but less accurate) than as_-distance_graph.

Examples:

```
list(ant) as_intersection_graph 0.5
```

See also:

as_distance_graph, as_edge_graph,

---

as_map

Possible use:

- `container as_map any expression` —> `map`
- `as_map (container , any expression)` —> `map`

Result:

produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand

Comment:

the right-hand operand should be a pair

Special cases:

- if the left-hand operand is nil, as_map throws an error.

Examples:

```
map<int,int> var0 <- [1,2,3,4,5,6,7,8] as_map (each::(each * 2));
      // var0 equals [1::2, 2::4, 3::6, 4::8, 5::10, 6::12,
   7::14, 8::16]
map<int,int> var1 <- [1::2,3::4,5::6] as_map (each::(each * 2));
      // var1 equals [2::4, 4::8, 6::12]
```

---

as_matrix

Possible use:

- `unknown` `as_matrix` `point` —> `matrix`
- `as_matrix` (`unknown` , `point`) —> `matrix`

Result:

casts the left operand into a matrix with right operand as preferred size

Comment:

This operator is very useful to cast a file containing raster data into a matrix.Note that both components of the right operand point should be positive, otherwise an exception is raised.The operator as_matrix creates a matrix of preferred size. It fills in it with elements of the left operand until the matrix is full If the size is to short, some elements will be omitted. Matrix remaining elements will be filled in by nil.

Special cases:

- if the right operand is nil, as_matrix is equivalent to the matrix operator

See also:

matrix,

---

as_path

Possible use:

- `list<geometry>` `as_path` `graph` —> `path`
- `as_path` (`list<geometry>` , `graph`) —> `path`

Result:

create a graph path from the list of shape

Examples:

```
path var0 <- [road1,road2,road3] as_path my_graph;  // var0
   equals a path road1->road2->road3 of my_graph
```

---

as_system_date

Possible use:

- `as_system_date` (`float`) —> `string`
- `float` `as_system_date` `string` —> `string`
- `as_system_date` (`float` , `string`) —> `string`

Result:

converts a number of milliseconds in the system (for instance, the 'machine_time' variable) into a string that represents the current date represented by these milliseconds using year, month, day, hour, minutes, seconds and time-zone offset following a given pattern (right-hand operand)

Comment:

as_system_date operator is a particular case (using a particular pattern) of the as_system_-date operator.

Special cases:

- Pattern should include : "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will return the complete date as defined by the ISO 8601 standard yyyy-MM-ddThh:mm:ss w/o the time-zone offset. Names are defined using the locale of the system

```
string var1 <- 2147483647 as_date " %D %Y %M / %h:%m:%s %z";
   // var1 equals "06 2015 05 / 23:58:57 +07"
```

Examples:

```
string var0 <- as_system_date(2147483647);  // var0 equals
   "2015-05-06"
```

See also:

as_system_date, as_system_time,

as_system_time

Possible use:

- as_system_time (float) —> string

Result:

converts a value of milliseconds in the system (for example, the value of the 'machine_time' variable) into a string representing the current hours, minutes and seconds in the current time zone of the machine and the current Locale. This representation follows the ISO 8601 standard hh:mm:ss

Comment:

as_system_time operator is a particular case (using a particular pattern) of the as_system_-date operator.

Examples:

```
string var0 <- as_system_time(2147483647);  // var0 equals
   "23:58:57"
```

See also:

as_system_date,

---

as_time

Possible use:

- as_time (float) —> string

Result:

converts a number of seconds in the model (for example, the value of the 'time' variable) into a string that represents the current number of hours, minutes and seconds of the period elapsed since the beginning of the simulation. As GAMA has no conception of time zones, the time is expressed as if the model was at GMT+00

Comment:

as_time operator is a particular case (using a particular pattern) of the as_date operator.

Examples:

```
string var0 <- as_time(22324234);   // var0 equals "09:10:34"
```

See also:

as_date,

---

asin

Possible use:

- `asin (int) —> float`
- `asin (float) —> float`

Result:

the arcsin of the operand

Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

Examples:

```
float var0 <- asin (90);     // var0 equals #nan
float var1 <- asin (0);      // var1 equals 0.0
```

See also:

acos, atan, sin,

`at`

Possible use:

- `string` `at` `int` —> `string`
- `at` (`string` , `int`) —> `string`
- `container<KeyType,ValueType>` `at` `KeyType` —> `ValueType`
- `at` (`container<KeyType,ValueType>` , `KeyType`) —> `ValueType`

Result:

the element at the right operand index of the container

Comment:

The first element of the container is located at the index 0. In addition, if the user tries to get the element at an index higher or equals than the length of the container, he will get an IndexOutOfBoundException.The at operator behavior depends on the nature of the operand

Special cases:

- if it is a file, at returns the element of the file content at the index specified by the right operand

- if it is a population, at returns the agent at the index specified by the right operand
- if it is a graph and if the right operand is a node, at returns the in and out edges corresponding to that node
- if it is a graph and if the right operand is an edge, at returns the pair node_out::node_-in of the edge
- if it is a graph and if the right operand is a pair node1::node2, at returns the edge from node1 to node2 in the graph
- if it is a list or a matrix, at returns the element at the index specified by the right operand

```
int var1 <- [1, 2, 3] at 2;      // var1 equals 3
point var2 <- [{1,2}, {3,4}, {5,6}] at 0;   // var2 equals
   {1.0,2.0}
```

Examples:

```
string var0 <- 'abcdef' at 0;   // var0 equals 'a'
```

See also:

contains_all, contains_any,

---

at_distance

Possible use:

- `container<agent> at_distance float —> list<geometry>`
- `at_distance (container<agent> , float) —> list<geometry>`

Result:

A list of agents or geometries among the left-operand list that are located at a distance <= the right operand from the caller agent (in its topology)

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] at_distance 20;  // var0
   equals the agents of the list located at a distance <= 20 from
    the caller agent (in the same order).
```

See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping,

---

at_location

Possible use:

- `geometry` `at_location` `point` —> `geometry`
- `at_location` (`geometry` , `point`) —> `geometry`

Result:

A geometry resulting from the tran of a translation to the right-hand operand point of the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self at_location {10, 20};     // var0 equals
   the geometry resulting from a translation to the location {10,
    20} of the left-hand geometry (or agent).
```

`atan`

Possible use:

- `atan` (`float`) —> `float`
- `atan` (`int`) —> `float`

Result:

Returns the value (in the interval [-90,90], in decimal degrees) of the arctan of the operand (which can be any real number).

Examples:

```
float var0 <- atan (1);      // var0 equals 45.0
```

See also:

acos, asin, tan,

`atan2`

Possible use:

- `float` `atan2` `float` —> `float`
- `atan2` (`float` , `float`) —> `float`

Result:

the atan2 value of the two operands.

Comment:

The function atan2 is the arctangent function with two arguments. The purpose of using two arguments instead of one is to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle, which is not possible for the single-argument arctangent function.

Examples:

```
float var0 <- atan2 (0,0);  // var0 equals 0.0
```

See also:

atan, acos, asin,

BDIPlan

Possible use:

- BDIPlan (any) —> BDIPlan

Result:

Casts the operand into the type BDIPlan

---

beta_index

Possible use:

- beta_index (graph) —> float

Result:

returns the beta index of the graph (Measures the level of connectivity in a graph and is expressed by the relationship between the number of links (e) over the number of nodes (v) : beta = e/v.

Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- beta_index(graphEpidemio);    // var1 equals the
    beta index of the graph
```

See also:

alpha_index, gamma_index, nb_cycles, connectivity_index,

`between`

Possible use:

- `between` (`float`, `float`, `float`) —> `bool`
- `between` (`int`, `int`, `int`) —> `bool`

Result:

returns true if the first float operand is bigger than the second float operand and smaller than the third float operand returns true the first integer operand is bigger than the second integer operand and smaller than the third integer operand

Examples:

```
bool var0 <- between(5.0, 1.0, 10.0);   // var0 equals true
bool var1 <- between(5, 1, 10);     // var1 equals true
```

---

betweenness_centrality

Possible use:

- `betweenness_centrality` (`graph`) —> `map`

Result:

returns a map containing for each vertex (key), its betweenness centrality (value): number of shortest paths passing through each vertex

Examples:

```
graph graphEpidemio <- graph([]);
map var1 <- betweenness_centrality(graphEpidemio);  // var1
   equals the betweenness centrality index of the graph
```

---

biggest_cliques_of

Possible use:

- `biggest_cliques_of` (`graph`) —> `list<list>`

Result:

returns the biggest cliques of a graph using the Bron-Kerbosch clique detection algorithm

Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- biggest_cliques_of (my_graph);   // var1
   equals the list of the biggest cliques as list
```

See also:

maximal_cliques_of,

`binomial`

Possible use:

- `int binomial float —> int`
- `binomial (int , float) —> int`

Result:

A value from a random variable following a binomial distribution. The operands represent the number of experiments n and the success probability p.

Comment:

The binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p, cf. Binomial distribution on Wikipedia.

Examples:

```
int var0 <- binomial(15,0.6);   // var0 equals a random positive
   integer
```

See also:

poisson, gauss,

`blend`

Possible use:

- `rgb` `blend` `rgb` —> `rgb`
- `blend` (`rgb` , `rgb`) —> `rgb`
- `blend` (`rgb`, `rgb`, `float`) —> `rgb`

Result:

Blend two colors with an optional ratio (c1 ∗ r + c2 ∗ (1 - r)) between 0 and 1

Special cases:

- If the ratio is omitted, an even blend is done

```
rgb var1 <- blend(#red, #blue);      // var1 equals to a color
   very close to the purple
```

Examples:

```
rgb var3 <- blend(#red, #blue, 0.3);     // var3 equals to a color
    between the purple and the blue
```

See also:

rgb, hsb,

`bool`

Possible use:

- `bool` (`any`) —> `bool`

Result:

Casts the operand into the type bool

`box`

Possible use:

- `box` (`point`) —> `geometry`
- `box` (`float`, `float`, `float`) —> `geometry`

Result:

A box geometry which side sizes are given by the operands.

Comment:

the center of the box is by default the location of the current agent in which has been called this operator.the center of the box is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- box(10, 5 , 5);    // var0 equals a geometry as
   a rectangle with width = 10, height = 5 depth= 5.
geometry var1 <- box({10, 5 , 5});  // var1 equals a geometry as
   a rectangle with width = 10, height = 5 depth= 5.
```

See also:

around, circle, sphere, cone, line, link, norm, point, polygon, polyline, square, cube, triangle,

---

`brewer_colors`

Possible use:

- `brewer_colors` (`string`) —> `list<rgb>`
- `string` `brewer_colors` `int` —> `list<rgb>`
- `brewer_colors` (`string` , `int`) —> `list<rgb>`

Result:

Build a list of colors of a given type (see website http://colorbrewer2.org/) with a given number of classes Build a list of colors of a given type (see website http://colorbrewer2.org/)

Examples:

```
list<rgb> var0 <- list<rgb> colors <- brewer_colors("Pastel1",
   10);;    // var0 equals a list of 10 sequential colors
list<rgb> var1 <- list<rgb> colors <- brewer_colors("OrRd");;
   // var1 equals a list of 6 blue colors
```

See also:

brewer_palettes,

---

brewer_palettes

Possible use:

- brewer_palettes (int) —> list<string>
- int brewer_palettes int —> list<string>
- brewer_palettes (int , int) —> list<string>

Result:

returns the list a palette with a given min number of classes and max number of classes) returns the list a palette with a given min number of classes and max number of classes)

Examples:

```
list<string> var0 <- list<rgb> colors <- brewer_palettes();;
   // var0 equals a list of palettes that are composed of a min
   of 5 colors
list<string> var1 <- list<rgb> colors <- brewer_palettes(5,10);;
      // var1 equals a list of palettes that are composed of a
   min of 5 colors and a max of 10 colors
```

See also:

brewer_colors,

---

buffer

Same signification as +

---

build

Possible use:

- build (matrix<float>) —> regression
- matrix<float> build string —> regression
- build (matrix<float> , string) —> regression

Result:

returns the regression build from the matrix data (a row = an instance, the last value of each line is the y value) while using the given method ("GLS" or "OLS"). Usage: build(data,method) returns the regression build from the matrix data (a row = an instance, the last value of each line is the y value) while using the given ordinary least squares method. Usage: build(data)

Examples:

```
build(matrix([[1,2,3,4],[2,3,4,2]]),"GLS")
matrix([[1,2,3,4],[2,3,4,2]])
```

---

ceil

Possible use:

- ceil (float) —> float

Result:

Maps the operand to the smallest following integer, i.e. the smallest integer not less than x.

Examples:

```
float var0 <- ceil(3);  // var0 equals 3.0
float var1 <- ceil(3.5);    // var1 equals 4.0
float var2 <- ceil(-4.7);   // var2 equals -4.0
```

See also:

floor, round,

---

change_clockwise

Possible use:

- `change_clockwise` (`geometry`) —> `geometry`

Result:

returns true if the geometry is defined clockwise

Examples:

```
geometry var0 <- is_clockwise(circle(10));  // var0 equals true
```

See also:

is_clockwise,

`char`

Possible use:

- `char` (`int`) —> `string`

Special cases:

- converts ACSII integer value to character

```
string var0 <- char (34);   // var0 equals '"'
```

`circle`

Possible use:

- `circle` (`float`) —> `geometry`
- `float` `circle` `point` —> `geometry`
- `circle` (`float` , `point`) —> `geometry`

Result:

A circle geometry which radius is equal to the operand. A circle geometry which radius is equal to the first operand, and the center has the location equal to the second operand.

Comment:

the center of the circle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.
- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- circle(10);     // var0 equals a geometry as a
    circle of radius 10.
geometry var1 <- circle(10,{80,30});     // var1 equals a geometry
     as a circle of radius 10, the center will be in the location
    {80,30}.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

clean

Possible use:

- `clean (geometry) —> geometry`

Result:

A geometry corresponding to the cleaning of the operand (geometry, agent, point)

Comment:

The cleaning corresponds to a buffer with a distance of 0.0

Examples:

```
geometry var0 <- clean(self);   // var0 equals returns the
   geometry resulting from the cleaning of the geometry of the
   agent applying the operator.
```

---

closest_points_with

Possible use:

- geometry closest_points_with geometry —> list<point>
- closest_points_with (geometry , geometry) —> list<point>

Result:

A list of two closest points between the two geometries.

Examples:

```
list<point> var0 <- geom1 closest_points_with(geom2);   // var0
   equals [pt1, pt2] with pt1 the closest point of geom1 to geom2
    and pt1 the closest point of geom2 to geom1
```

See also:

any_location_in, any_point_in, farthest_point_to, points_at,

`closest_to`

Possible use:

- `container<agent>` `closest_to` `geometry` —> `geometry`
- `closest_to` (`container<agent>` , `geometry`) —> `geometry`

Result:

An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the closest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
geometry var0 <- [ag1, ag2, ag3] closest_to(self);  // var0
   equals return the closest agent among ag1, ag2 and ag3 to the
   agent applying the operator.
(species1 + species2) closest_to self
```

See also:

neighbors_at, neighbors_of, inside, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

`collect`

Possible use:

- `container` `collect` `any expression` —> `container`
- `collect` (`container` , `any expression`) —> `container`

Result:

returns a new list, in which each element is the evaluation of the right-hand operand.

Comment:

collect is similar to accumulate except that accumulate always produces flat lists if the right-hand operand returns a list.In addition, collect can be applied to any container.

Special cases:

- if the left-hand operand is nil, collect throws an error

Examples:

```
container var0 <- [1,2,4] collect (each *2);    // var0 equals
   [2,4,8]
container var1 <- [1,2,4] collect ([2,4]);  // var1 equals
   [[2,4],[2,4],[2,4]]
container var2 <- [1::2, 3::4, 5::6] collect (each + 2);    //
   var2 equals [4,6,8]
container var3 <- (list(node) collect (node(each).location.x * 2)
   ;  // var3 equals the list of nodes with their x multiplied by
    2
```

See also:

accumulate,

---

column_at

Possible use:

- `matrix column_at int —> list`
- `column_at (matrix , int) —> list`

Result:

returns the column at a num_col (right-hand operand)

Examples:

```
list var0 <- matrix([["el11","el12","el13"],["el21","el22","el23"
   ],["el31","el32","el33"]]) column_at 2;    // var0 equals ["
   el31","el32","el33"]
```

See also:

row_at, rows_list,

---

columns_list

Possible use:

- `columns_list` (`matrix`) —> `list<list>`

Result:

returns a list of the columns of the matrix, with each column as a list of elements

Examples:

```
list<list> var0 <- columns_list(matrix([["el11","el12","el13"],["
   el21","el22","el23"],["el31","el32","el33"]]));    // var0
   equals [["el11","el12","el13"],["el21","el22","el23"],["el31
   ","el32","el33"]]
```

See also:

rows_list,

`command`

Possible use:

- `command` (`string`) —> `string`

Result:

command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence '&' is used at the end. In this case, the result of the operator is an empty string

---

`cone`

Possible use:

- `cone` (`point`) —> `geometry`
- `int` `cone` `int` —> `geometry`
- `cone` (`int` , `int`) —> `geometry`

Result:

A cone geometry which min and max angles are given by the operands. A cone geometry which min and max angles are given by the operands.

Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.the center of the cone is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- cone({0, 45});      // var0 equals a geometry as
   a cone with min angle is 0 and max angle is 45.
geometry var1 <- cone(0, 45);    // var1 equals a geometry as a
   cone with min angle is 0 and max angle is 45.
```

See also:

around, circle, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

cone3D

Possible use:

- `float` `cone3D` `float` —> `geometry`
- `cone3D` (`float` , `float`) —> `geometry`

Result:

A cone geometry which radius is equal to the operand.

Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- cone3D(10.0,10.0);      // var0 equals a geometry
    as a circle of radius 10 but displays a cone.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

connected_components_of

Possible use:

- `connected_components_of` (`graph`) —> `list<list>`

Result:

returns the connected components of of a graph, i.e. the list of all vertices that are in the maximally connected component together with the specified vertex.

Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- connected_components_of (my_graph);  // var1
   equals the list of all the components as list
```

See also:

alpha_index, connectivity_index, nb_cycles,

---

connectivity_index

Possible use:

- `connectivity_index` (`graph`) —> `float`

Result:

returns a simple connectivity index. This number is estimated through the number of nodes (v) and of sub-graphs (p) : IC = (v - p) /(v - 1).

Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- connectivity_index(graphEpidemio);    // var1
   equals the connectivity index of the graph
```

See also:

alpha_index, beta_index, gamma_index, nb_cycles,

`container`

Possible use:

- `container` (`any`) —> `container`

Result:

Casts the operand into the type container

`contains`

Possible use:

- `string` `contains` `string` —> `bool`
- `contains` (`string` , `string`) —> `bool`
- `container<KeyType,ValueType>` `contains` `unknown` —> `bool`
- `contains` (`container<KeyType,ValueType>` , `unknown`) —> `bool`

Result:

true, if the container contains the right operand, false otherwise

Comment:

the contains operator behavior depends on the nature of the operand

Special cases:

- if both operands are strings, returns true if the right-hand operand contains the right-hand pattern;

- if it is a map, contains returns true if the operand is a key of the map

- if it is a file, contains returns true it the operand is contained in the file content
- if it is a population, contains returns true if the operand is an agent of the population, false otherwise
- if it is a graph, contains returns true if the operand is a node or an edge of the graph, false otherwise
- if it is a list or a matrix, contains returns true if the list or matrix contains the right operand

```
bool var1 <- [1, 2, 3] contains 2;  // var1 equals true
bool var2 <- [{1,2}, {3,4}, {5,6}] contains {3,4};  // var2
   equals true
```

Examples:

```
bool var0 <- 'abcded' contains 'bc';    // var0 equals true
```

See also:

contains_all, contains_any,

---

contains_all

Possible use:

- `container` `contains_all` `container` —> `bool`
- `contains_all` (`container` , `container`) —> `bool`
- `string` `contains_all` `list` —> `bool`
- `contains_all` (`string` , `list`) —> `bool`

Result:

true if the left operand contains all the elements of the right operand, false otherwise

Comment:

the definition of contains depends on the container

Special cases:

- if the right operand is nil or empty, contains_all returns true
- if the left-operand is a string, test whether the string contains all the element of the list;

```
bool var4 <- "abcabcabc" contains_all ["ca","xy"];  // var4
   equals false
```

Examples:

```
bool var0 <- [1,2,3,4,5,6] contains_all [2,4];  // var0 equals
   true
bool var1 <- [1,2,3,4,5,6] contains_all [2,8];  // var1 equals
   false
bool var2 <- [1::2, 3::4, 5::6] contains_all [1,3];     // var2
   equals false
bool var3 <- [1::2, 3::4, 5::6] contains_all [2,4];     // var3
   equals true
```

See also:

contains, contains_any,

contains_any

Possible use:

- string contains_any list —> bool
- contains_any (string , list) —> bool
- container contains_any container —> bool
- contains_any (container , container) —> bool

Result:

true if the left operand contains one of the elements of the right operand, false otherwise

Comment:

the definition of contains depends on the container

Special cases:

- if the right operand is nil or empty, contains_any returns false

Examples:

```
bool var0 <- "abcabcabc" contains_any ["ca","xy"];  // var0
   equals true
bool var1 <- [1,2,3,4,5,6] contains_any [2,4];  // var1 equals
   true
bool var2 <- [1,2,3,4,5,6] contains_any [2,8];  // var2 equals
   true
bool var3 <- [1::2, 3::4, 5::6] contains_any [1,3];     // var3
   equals false
bool var4 <- [1::2, 3::4, 5::6] contains_any [2,4];     // var4
   equals true
```

See also:

contains, contains_all,

---

contains_edge

Possible use:

- graph contains_edge unknown —> bool
- contains_edge (graph , unknown) —> bool
- graph contains_edge pair —> bool
- contains_edge (graph , pair) —> bool

Result:

returns true if the graph(left-hand operand) contains the given edge (righ-hand operand), false otherwise

Special cases:

- if the left-hand operand is nil, returns false

- if the right-hand operand is a pair, returns true if it exists an edge between the two elements of the pair in the graph

```
bool var2 <- graphEpidemio contains_edge (node(0)::node(3));
   // var2 equals true
```

Examples:

```
graph graphFromMap <-  as_edge_graph
   ([{1,5}::{12,45},{12,45}::{34,56}]);
bool var1 <- graphFromMap contains_edge link({1,5}::{12,45});
   // var1 equals true
```

See also:

contains_vertex,

---

contains_vertex

Possible use:

- graph contains_vertex unknown —> bool
- contains_vertex (graph , unknown) —> bool

Result:

returns true if the graph(left-hand operand) contains the given vertex (righ-hand operand), false otherwise

Special cases:

- if the left-hand operand is nil, returns false

Examples:

```
graph graphFromMap<-  as_edge_graph
   ([{1,5}::{12,45},{12,45}::{34,56}]);
bool var1 <- graphFromMap contains_vertex {1,5};    // var1
   equals true
```

See also:

contains_edge,

---

conversation

Possible use:

- `conversation` (`unknown`) —> `conversation`

---

convex_hull

Possible use:

- `convex_hull` (`geometry`) —> `geometry`

Result:

A geometry corresponding to the convex hull of the operand.

Examples:

```
geometry var0 <- convex_hull(self);      // var0 equals the convex
    hull of the geometry of the agent applying the operator
```

---

copy

Possible use:

- `copy` (`unknown`) —> `unknown`

Result:

returns a copy of the operand.

---

copy_between

Possible use:

- copy_between (string, int, int) —> string
- copy_between (container, int, int) —> container

Result:

Returns a copy of the first operand between the indexes determined by the second (inclusive) and third operands (exclusive)

Special cases:

- If the first operand is empty, returns an empty object of the same type
- If the second operand is greater than or equal to the third operand, return an empty object of the same type
- If the first operand is nil, raises an error

Examples:

```
string var0 <- copy_between("abcabcabc", 2,6);  // var0 equals "
   cabc"
container var1 <-  copy_between ([4, 1, 6, 9 ,7], 1, 3);    //
   var1 equals [1, 6]
```

---

corR

Possible use:

- container corR container —> unknown
- corR (container , container) —> unknown

Result:

returns the Pearson correlation coefficient of two given vectors (right-hand operands) in given variable (left-hand operand).

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
list X <- [1, 2, 3];
list Y <- [1, 2, 4];
unknown var2 <- corR(X, Y);      // var2 equals 0.981980506061966
```

---

cos

Possible use:

- `cos` (`float`) —> `float`
- `cos` (`int`) —> `float`

Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- cos (0);  // var0 equals 1.0
float var1 <- cos(360);     // var1 equals 1.0
float var2 <- cos(-720);    // var2 equals 1.0
```

See also:

sin, tan,

---

cos_rad

Possible use:

- `cos_rad` (`float`) —> `float`

Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

See also:

sin, tan,

---

count

Possible use:

- container count any expression —> int
- count (container , any expression) —> int

Result:

returns an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

Special cases:

- if the left-hand operand is nil, count throws an error

Examples:

```
int var0 <- [1,2,3,4,5,6,7,8] count (each > 3);     // var0
   equals 5
// Number of nodes of graph g2 without any out edge
graph g2 <- graph([]);
int var3 <- g2 count (length(g2 out_edges_of each) = 0  ) ;
   // var3 equals the total number of out edges
// Number of agents node with x > 32
int n <- (list(node) count (round(node(each).location.x) > 32);
int var6 <- [1::2, 3::4, 5::6] count (each > 4);    // var6
   equals 1
```

See also:

group_by,

---

covers

Possible use:

- `geometry` `covers` `geometry` —> `bool`
- `covers` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) covers the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- square(5) covers square(2);    // var0 equals true
```

See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

cross

Possible use:

- cross (float) —> geometry
- float cross float —> geometry
- cross (float , float) —> geometry

Result:

A cross, which radius is equal to the first operand and the width of the lines for the second
A cross, which radius is equal to the first operand

Examples:

```
geometry var0 <- cross(10,2);   // var0 equals a geometry as a
   cross of radius 10, and with a width of 2 for the lines
geometry var1 <- cross(10);     // var1 equals a geometry as a
   cross of radius 10
```

See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

crosses

Possible use:

- geometry crosses geometry —> bool
- crosses (geometry , geometry) —> bool

Result:

A boolean, equal to true if the left-geometry (or agent/point) crosses the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.
- if one operand is a point, returns false.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) crosses polyline
   ([{10,20},{20,10}]);   // var0 equals true
bool var1 <- polyline([{10,10},{20,20}]) crosses {15,15};   //
   var1 equals true
bool var2 <- polyline([{0,0},{25,25}]) crosses polygon
   ([{10,10},{10,20},{20,20},{20,10}]);  // var2 equals true
```

See also:

disjoint_from, intersects, overlaps, partially_overlaps, touches,

---

crs

Possible use:

- crs (file) —> string

Result:

the Coordinate Reference System (CRS) of the GIS file

Examples:

```
string var0 <- crs(my_shapefile);   // var0 equals the crs of the
    shapefile
```

CRS_transform

Possible use:

- CRS_transform (geometry) —> geometry
- geometry CRS_transform string —> geometry
- CRS_transform (geometry , string) —> geometry

Special cases:

- returns the geometry corresponding to the transformation of the given geometry by the left operand CRS (Coordinate Reference System)

```
geometry var0 <- shape CRS_transform("EPSG:4326");  // var0
   equals a geometry corresponding to the agent geometry
   transformed into the EPSG:4326 CRS
```

- returns the geometry corresponding to the transformation of the given geometry by the current CRS (Coordinate Reference System), the one corresponding to the world's agent one

```
geometry var1 <- CRS_transform(shape);  // var1 equals a geometry
    corresponding to the agent geometry transformed into the
   current CRS
```

---

csv_file

Possible use:

- `csv_file` (`string`) —> `file`

Result:

Constructs a file of type csv. Allowed extensions are limited to csv, tsv

---

cube

Possible use:

- `cube` (`float`) —> `geometry`

Result:

A cube geometry which side size is equal to the operand.

Comment:

the center of the cube is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- cube(10);  // var0 equals a geometry as a square
    of side size 10.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

`curve`

Possible use:

- `curve` (`point`, `point`, `point`) —> `geometry`
- `curve` (`point`, `point`, `point`, `int`) —> `geometry`
- `curve` (`point`, `point`, `point`, `point`) —> `geometry`
- `curve` (`point`, `point`, `point`, `point`, `int`) —> `geometry`

Result:

A cubic Bezier curve geometry built from the four given points composed of a given number of points. A quadratic Bezier curve geometry built from the three given points composed of 10 points. A quadratic Bezier curve geometry built from the three given points composed of a given numnber of points. A cubic Bezier curve geometry built from the four given points composed of 10 points.

Special cases:

- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil

Examples:

```
geometry var0 <- curve({0,0}, {0,10}, {10,10});     // var0
   equals a cubic Bezier curve geometry composed of 10 points
   from p0 to p3.
geometry var1 <- curve({0,0}, {0,10}, {10,10});     // var1
   equals a quadratic Bezier curve geometry composed of 10 points
    from p0 to p2.
geometry var2 <- curve({0,0}, {0,10}, {10,10}, 20);     // var2
   equals a quadratic Bezier curve geometry composed of 20 points
    from p0 to p2.
geometry var3 <- curve({0,0}, {0,10}, {10,10});     // var3
   equals a cubic Bezier curve geometry composed of 10 points
   from p0 to p3.
```

See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle, line,

---

cylinder

Possible use:

- `float` `cylinder` `float` —> `geometry`
- `cylinder` (`float` , `float`) —> `geometry`

Result:

A cylinder geometry which radius is equal to the operand.

Comment:

the center of the cylinder is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- cylinder(10,10);   // var0 equals a geometry as
    a circle of radius 10.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

`date`

Possible use:

- `date` (`any`) —> `date`

Result:

Casts the operand into the type date

`dbscan`

Possible use:

- `dbscan` (`list`, `float`, `int`) —> `list<list>`

Result:

returns the list of clusters (list of instance indices) computed with the dbscan (density-based spatial clustering of applications with noise) algorithm from the first operand data according to the maximum radius of the neighborhood to be considered (eps) and the minimum number of points needed for a cluster (minPts). Usage: dbscan(data,eps,minPoints)

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
dbscan ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],10,2)
```

---

dead

Possible use:

- `dead (agent) —> bool`

Result:

true if the agent is dead (or null), false otherwise.

Examples:

```
bool var0 <- dead(agent_A);     // var0 equals true or false
```

`degree_of`

Possible use:

- `graph` `degree_of` `unknown` —> `int`
- `degree_of` (`graph` , `unknown`) —> `int`

Result:

returns the degree (in+out) of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap degree_of (node(3));   // var1 equals 3
```

See also:

in_degree_of, out_degree_of,

`dem`

Possible use:

- `dem` (`file`) —> `geometry`
- `file` `dem` `float` —> `geometry`
- `dem` (`file` , `float`) —> `geometry`
- `file` `dem` `file` —> `geometry`
- `dem` (`file` , `file`) —> `geometry`
- `dem` (`file`, `file`, `float`) —> `geometry`

Result:

A polygon that is equivalent to the surface of the texture

Examples:

```
geometry var0 <- dem(dem,z_factor);     // var0 equals a geometry
     as a rectangle of weight and height equal to the texture.
geometry var1 <- dem(dem,texture,z_factor);     // var1 equals a
    geometry as a rectangle of width and height equal to the
    texture.
geometry var2 <- dem(dem);  // var2 equals returns a geometry as
    a rectangle of width and height equal to the texture.
geometry var3 <- dem(dem,texture);  // var3 equals a geometry as
    a rectangle of weight and height equal to the texture.
```

---

det

Same signification as determinant

---

determinant

Possible use:

- `determinant` (`matrix`) —> `float`

Result:

The determinant of the given matrix

Examples:

```
float var0 <- determinant(matrix([[1,2],[3,4]]));   // var0
    equals -2
```

`diff`

Possible use:

- `float` `diff` `float` —> `float`
- `diff` (`float` , `float`) —> `float`

`diff2`

Possible use:

- `float` `diff2` `float` —> `float`
- `diff2` (`float` , `float`) —> `float`

`directed`

Possible use:

- `directed` (`graph`) —> `graph`

Result:

the operand graph becomes a directed graph.

Comment:

the operator alters the operand graph, it does not create a new one.

See also:

undirected,

`direction_between`

Possible use:

- `topology` `direction_between` `container<geometry>` —> `int`
- `direction_between` (`topology` , `container<geometry>`) —> `int`

Result:

A direction (in degree) between a list of two geometries (geometries, agents, points) considering a topology.

Examples:

```
int var0 <- my_topology direction_between [ag1, ag2];    // var0
   equals the direction between ag1 and ag2 considering the
   topology my_topology
```

See also:

towards, direction_to, distance_to, distance_between, path_between, path_to,

---

`direction_to`

Same signification as towards

`disjoint_from`

Possible use:

- `geometry` `disjoint_from` `geometry` —> `bool`
- `disjoint_from` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) is disjoints from the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns true.
- if one operand is a point, returns false if the point is included in the geometry.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) disjoint_from polyline
   ([{15,15},{25,25}]);     // var0 equals false
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   disjoint_from polygon([{15,15},{15,25},{25,25},{25,15}]);   //
    var1 equals false
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   disjoint_from {15,15};  // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   disjoint_from {25,25};  // var3 equals true
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   disjoint_from polygon([{35,35},{35,45},{45,45},{45,35}]);   //
    var4 equals true
```

See also:

intersects, crosses, overlaps, partially_overlaps, touches,

`distance_between`

Possible use:

- `topology` **`distance_between`** `container<geometry>` —> `float`

- distance_between (topology , container<geometry>) —> float

Result:

A distance between a list of geometries (geometries, agents, points) considering a topology.

Examples:

```
float var0 <- my_topology distance_between [ag1, ag2, ag3];
   // var0 equals the distance between ag1, ag2 and ag3
   considering the topology my_topology
```

See also:

towards, direction_to, distance_to, direction_between, path_between, path_to,

---

distance_to

Possible use:

- point distance_to point —> float
- distance_to (point , point) —> float
- geometry distance_to geometry —> float
- distance_to (geometry , geometry) —> float

Result:

A distance between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

Examples:

```
float var0 <- ag1 distance_to ag2;  // var0 equals the distance
   between ag1 and ag2 considering the topology of the agent
   applying the operator
```

See also:

towards, direction_to, distance_between, direction_between, path_between, path_to,

---

distribution_of

Possible use:

- `distribution_of` (`container`) —> `map`
- `container` `distribution_of` `int` —> `map`
- `distribution_of` (`container` , `int`) —> `map`
- `distribution_of` (`container`, `int`, `float`, `float`) —> `map`

Result:

Discretize a list of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nbbins) or (list,nbbins,valmin,valmax)

Examples:

```
map var0 <- distribution_of([1,1,2,12.5]);  // var0 equals map(['
   values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12
   parlist'::[1,0]])
```

```
map var1 <- distribution_of([1,1,2,12.5],10);   // var1 equals
   map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]',
   parlist'::[1,0]])
map var2 <- distribution_of([1,1,2,12.5]);  // var2 equals map(['
   values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]',
   parlist'::[1,0]])
```

See also:

as_map,

`distribution2d_of`

Possible use:

- `container` **`distribution2d_of`** `container` —> `map`
- **`distribution2d_of`** (`container` , `container`) —> `map`
- **`distribution2d_of`** (`container`, `container`, `int`, `int`) —> `map`
- **`distribution2d_of`** (`container`, `container`, `int`, `float`, `float`, `int`, `float`, `float`) —> `map`

Result:

Discretize two lists of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nbbins) or (list,nbbins,valmin,valmax)

Examples:

```
map var0 <- distribution_of([1,1,2,12.5],10);   // var0 equals
   map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12
   parlist'::[1,0]])
map var1 <- distribution2d_of([1,1,2,12.5]);    // var1 equals
   map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12
   parlist'::[1,0]])
map var2 <- distribution_of([1,1,2,12.5],10);   // var2 equals
   map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend
   '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12
   parlist'::[1,0]])
```

See also:

as_map,

---

div

Possible use:

- `float div int —> int`
- `div (float , int) —> int`
- `int div float —> int`
- `div (int , float) —> int`
- `float div float —> int`
- `div (float , float) —> int`
- `int div int —> int`
- `div (int , int) —> int`

Result:

Returns the truncation of the division of the left-hand operand by the right-hand operand.

Special cases:

- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.

Examples:

```
int var0 <- 40.5 div 3;      // var0 equals 13
int var1 <- 40 div 4.1;      // var1 equals 9
int var2 <- 40.1 div 4.5;    // var2 equals 8
int var3 <- 40 div 3;    // var3 equals 13
```

See also:

mod,

---

dxf_file

Possible use:

- `dxf_file` (`string`) —> `file`

Result:

Constructs a file of type dxf. Allowed extensions are limited to dxf

---

edge

Possible use:

- edge (unknown) —> unknown
- edge (pair) —> unknown
- unknown edge float —> unknown
- edge (unknown , float) —> unknown
- pair edge float —> unknown
- edge (pair , float) —> unknown
- unknown edge unknown —> unknown
- edge (unknown , unknown) —> unknown
- edge (pair, unknown, float) —> unknown
- edge (unknown, unknown, unknown) —> unknown
- edge (unknown, unknown, float) —> unknown
- edge (unknown, unknown, unknown, float) —> unknown

---

edge_between

Possible use:

- graph edge_between pair —> unknown
- edge_between (graph , pair) —> unknown

Result:

returns the edge linking two nodes

Examples:

```
unknown var0 <- graphFromMap edge_between node1::node2;     //
   var0 equals edge1
```

See also:

out_edges_of, in_edges_of,

---

edges

Possible use:

- `edges (container) ―> container`

---

eigenvalues

Possible use:

- `eigenvalues (matrix) ―> list<float>`

Result:

The eigen values (matrix) of the given matrix

Examples:

```
list<float> var0 <- eigenvalues(matrix([[5,-3],[6,-4]]));    //
   var0 equals [2.0000000000000004,-0.9999999999999998]
```

---

electre_DM

Possible use:

- `electre_DM (list<list>, list<map<string,object>>, float) ―> int`

Result:

The index of the best candidate according to a method based on the ELECTRE methods. The principle of the ELECTRE methods is to compare the possible candidates by pair. These methods analyses the possible outranking relation existing between two candidates. An candidate outranks another if this one is at least as good as the other one. The ELECTRE methods are based on two concepts: the concordance and the discordance. The concordance characterizes the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterizes the fact that, for an outranking relation to be validated, none of the criteria in the minority should oppose too strongly this assertion. These two conditions must be true for validating the outranking assertion. More information about the ELECTRE methods can be found in [http://www.springerlink.com/content/g367r44322876223/ Figueira, J., Mousseau, V., Roy, B.: ELECTRE Methods. In: Figueira, J., Greco, S., and Ehrgott, M., (Eds.), Multiple Criteria Decision Analysis: State of the Art Surveys, Springer, New York, 133–162 (2005)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fives elements: a name, a weight, a preference value (p), an indifference value (q) and a veto value (v). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant. The veto value represents the threshold from which the difference between two criterion values disqualifies the candidate that obtained the smaller value; the last operand is the fuzzy cut.

Special cases:

- returns -1 is the list of candidates is nil or empty

Examples:

```
int var0 <- electre_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name
   "::"utility", "weight" :: 2.0,"p"::0.5, "q"::0.0, "s"::1.0, "
   maximize" :: true],["name"::"price", "weight" :: 1.0,"p"::0.5,
    "q"::0.0, "s"::1.0, "maximize" :: false]]);     // var0 equals
    0
```

See also:

weighted_means_DM, promethee_DM, evidence_theory_DM,

`ellipse`

Possible use:

- `float` `ellipse` `float` —> `geometry`
- `ellipse` (`float` , `float`) —> `geometry`

Result:

An ellipse geometry which x-radius is equal to the first operand and y-radius is equal to the second operand

Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if both operands are lower or equal to 0, a line if only one is.

Examples:

```
geometry var0 <- ellipse(10, 10);   // var0 equals a geometry as
   an ellipse of width 10 and height 10.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, circle, squircle, triangle,

---

`emotion`

Possible use:

- `emotion` (`any`) —> `emotion`

Result:

Casts the operand into the type emotion

`empty`

Possible use:

- `empty` (`string`) —> `bool`
- `empty` (`container<KeyType,ValueType>`) —> `bool`

Result:

true if the operand is empty, false otherwise.

Comment:

the empty operator behavior depends on the nature of the operand

Special cases:

- if it is a map, empty returns true if the map contains no key-value mappings, and false otherwise
- if it is a file, empty returns true if the content of the file (that is also a container) is empty, and false otherwise
- if it is a population, empty returns true if there is no agent in the population, and false otherwise
- if it is a graph, empty returns true if it contains no vertex and no edge, and false otherwise
- if it is a matrix of int, float or object, it will return true if all elements are respectively 0, 0.0 or null, and false otherwise
- if it is a matrix of geometry, it will return true if the matrix contains no cell, and false otherwise
- if it is a string, empty returns true if the string does not contain any character, and false otherwise

```
bool var0 <- empty ('abced');   // var0 equals false
```

- if it is a list, empty returns true if there is no element in the list, and false otherwise

```
bool var1 <- empty([]);     // var1 equals true
```

`enlarged_by`

Same signification as +

---

envelope

Possible use:

- `envelope` (`unknown`) —> `geometry`

Result:

A 3D geometry that represents the box that surrounds the geometries or the surface described by the arguments. More general than geometry(arguments).envelope, as it allows to pass int, double, point, image files, shape files, asc files, or any list combining these arguments, in which case the envelope will be correctly expanded. If an envelope cannot be determined from the arguments, a default one of dimensions (0,100, 0, 100, 0, 100) is returned

---

eval_gaml

Possible use:

- `eval_gaml` (`string`) —> `unknown`

Result:

evaluates the given GAML string.

Examples:

```
unknown var0 <- eval_gaml("2+3");   // var0 equals 5
```

---

eval_when

Possible use:

- `eval_when (BDIPlan) —> bool`

Result:

evaluate the facet when of a given plan

Examples:

```
eval_when(plan1)
```

---

even

Possible use:

- `even (int) —> bool`

Result:

Returns true if the operand is even and false if it is odd.

Special cases:

- if the operand is equal to 0, it returns true.
- if the operand is a float, it is truncated before

Examples:

```
bool var0 <- even (3);  // var0 equals false
bool var1 <- even(-12);     // var1 equals true
```

---

every

Possible use:

- every (int) —> bool
- container every int —> container
- every (container , int) —> container

Result:

Retrieves elements from the first argument every step (second argument) elements. Raises an error if the step is negative or equal to zero true every operand * cycle, false otherwise

Comment:

the value of the every operator depends on the cycle. It can be used to do something every x cycle.

Examples:

```
if every(2) {write "the time step is even";}
         else {write "the time step is odd";}
```

`evidence_theory_DM`

Possible use:

- `list<list>` `evidence_theory_DM` `list<map<string,object>>` —> `int`
- `evidence_theory_DM` (`list<list>` , `list<map<string,object>>`) —> `int`
- `evidence_theory_DM` (`list<list>`, `list<map<string,object>>`, `bool`) —> `int`

Result:

The index of the best candidate according to a method based on the Evidence theory. This theory, which was proposed by Shafer ([http://www.glennshafer.com/books/amte.html Shafer G (1976) A mathematical theory of evidence, Princeton University Press]), is based on the work of Dempster ([http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177698950 Dempster A (1967) Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, vol. 38, pp. 325–339]) on lower and upper probability distributions. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains seven elements: a name, a first threshold s1, a second threshold s2, a value for the assertion "this candidate is the best" at threshold s1 (v1p), a value for the assertion "this candidate is the best" at threshold s2 (v2p), a value for the assertion "this candidate is not the best" at threshold s1 (v1c), a value for the assertion "this candidate is not the best" at threshold s2 (v2c). v1p, v2p, v1c and v2c have to been defined in order that: v1p + v1c <= 1.0; v2p + v2c <= 1.0.; the last operand allows to use a simple version of this multi-criteria decision making method (simple if true)

Special cases:

- returns -1 is the list of candidates is nil or empty
- if the operator is used with only 2 operands (the candidates and the criteria), the last parameter (use simple method) is set to true

Examples:

```
int var0 <- evidence_theory_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]],
    [["name"::"utility", "s1" :: 0.0,"s2"::1.0, "v1p"::0.0, "v2p"
   ::1.0, "v1c"::0.0, "v2c"::0.0, "maximize" :: true],["name"::"
   price",  "s1" :: 0.0,"s2"::1.0, "v1p"::0.0, "v2p"::1.0, "v1c"
   ::0.0, "v2c"::0.0, "maximize" :: true]], true);    // var0
   equals 0
```

See also:

weighted_means_DM, electre_DM,

---

exp

Possible use:

- `exp` (`float`) —> `float`
- `exp` (`int`) —> `float`

Result:

Returns Euler's number e raised to the power of the operand.

Special cases:

- the operand is casted to a float before being evaluated.
- the operand is casted to a float before being evaluated.

Examples:

```
float var0 <- exp (0);  // var0 equals 1.0
```

See also:

ln,

`fact`

Possible use:

- `fact (int) —> float`

Result:

Returns the factorial of the operand.

Special cases:

- if the operand is less than 0, fact returns 0.

Examples:

```
float var0 <- fact(4);  // var0 equals 24
```

`farthest_point_to`

Possible use:

- `geometry farthest_point_to point —> point`
- `farthest_point_to (geometry , point) —> point`

Result:

the farthest point of the left-operand to the left-point.

Examples:

```
point var0 <- geom farthest_point_to(pt);   // var0 equals the
   farthest point of geom to pt
```

See also:

any_location_in, any_point_in, closest_points_with, points_at,

---

`farthest_to`

Possible use:

- `container<agent>` `farthest_to` `geometry` —> `geometry`
- `farthest_to` (`container<agent>` , `geometry`) —> `geometry`

Result:

An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the farthest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
geometry var0 <- [ag1, ag2, ag3] closest_to(self);  // var0
   equals return the farthest agent among ag1, ag2 and ag3 to the
    agent applying the operator.
(species1 + species2) closest_to self
```

See also:

neighbors_at, neighbors_of, inside, overlapping, agents_overlapping, agents_inside, agent_closest_to, closest_to, agent_farthest_to,

---

file

Possible use:

- `file` (`string`) —> `file`
- `string` `file` `container` —> `file`
- `file` (`string` , `container`) —> `file`

Result:

opens a file in read only mode, creates a GAML file object, and tries to determine and store the file content in the contents attribute. Creates a file in read/write mode, setting its contents to the container passed in parameter

Comment:

The file should have a supported extension, see file type definition for supported file extensions.The type of container to pass will depend on the type of file (see the management of files in the documentation). Can be used to copy files since files are considered as containers. For example: save file('image_copy.png', file('image.png')); will copy image.png to image_copy.png

Special cases:

- If the specified string does not refer to an existing file, an exception is risen when the variable is used.

Examples:

```
let fileT type: file value: file("../includes/Stupid_Cell.Data");
            // fileT represents the file "../includes/Stupid_Cell
   .Data"
            // fileT.contents here contains a matrix storing all
   the data of the text file
```

See also:

folder, new_folder,

`file_exists`

Possible use:

- `file_exists` (`string`) —> `bool`

Result:

Test whether the parameter is the path to an existing file.

`first`

Possible use:

- `first` (`string`) —> `string`
- `first` (`container<KeyType,ValueType>`) —> `ValueType`
- `int` `first` `container` —> `container`
- `first` (`int` , `container`) —> `container`

Result:

the first value of the operand

Comment:

the first operator behavior depends on the nature of the operand

Special cases:

- if it is a map, first returns the first value of the first pair (in insertion order)
- if it is a file, first returns the first element of the content of the file (that is also a container)
- if it is a population, first returns the first agent of the population
- if it is a graph, first returns the first edge (in creation order)
- if it is a matrix, first returns the element at {0,0} in the matrix
- for a matrix of int or float, it will return 0 if the matrix is empty
- for a matrix of object or geometry, it will return nil if the matrix is empty
- if it is a string, first returns a string composed of its first character

```
string var0 <- first ('abce');  // var0 equals 'a'
```

- if it is a list, first returns the first element of the list, or nil if the list is empty

```
int var1 <- first ([1, 2, 3]);  // var1 equals 1
```

See also:

last,

`first_with`

Possible use:

- `container` `first_with` `any expression` —> `unknown`
- `first_with` (`container` , `any expression`) —> `unknown`

Result:

the first element of the left-hand operand that makes the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, first_with throws an error. If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] first_with (each >= 4);  //
   var4 equals 4
unknown var5 <- [1::2, 3::4, 5::6].pairs first_with (each.value
   >= 4);  // var5 equals 3::4
```

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] first_with (each > 3);    //
   var0 equals 4
unknown var2 <- g2 first_with (length(g2 out_edges_of each) = 0);
      // var2 equals node9
unknown var3 <- (list(node) first_with (round(node(each).location
   .x) > 32);     // var3 equals node2
```

See also:

group_by, last_with, where,

---

flip

Possible use:

- flip (float) —> bool

Result:

true or false given the probability represented by the operand

Special cases:

- flip 0 always returns false, flip 1 true

Examples:

```
bool var0 <- flip (0.66666);     // var0 equals 2/3 chances to
   return true.
```

See also:

rnd,

---

float

Possible use:

- `float` (`any`) —> `float`

Result:

Casts the operand into the type float

---

floor

Possible use:

- `floor` (`float`) —> `float`

Result:

Maps the operand to the largest previous following integer, i.e. the largest integer not greater than x.

Examples:

```
float var0 <- floor(3);     // var0 equals 3.0
float var1 <- floor(3.5);   // var1 equals 3.0
float var2 <- floor(-4.7);  // var2 equals -5.0
```

See also:

ceil, round,

`folder`

Possible use:

- `folder` (`string`) —> `file`

Result:

opens an existing repository

Special cases:

- If the specified string does not refer to an existing repository, an exception is risen.

Examples:

```
folder("../includes/")
file dirT <- folder("../includes/");
                // dirT represents the repository "../includes/"
                // dirT.contents here contains the list of the
   names of included files
```

See also:

file, new_folder,

---

font

Possible use:

- `font` (`string`, `int`, `int`) —> `font`

Result:

Creates a new font, by specifying its name (either a font face name like 'Lucida Grande Bold' or 'Helvetica', or a logical name like 'Dialog', 'SansSerif', 'Serif', etc.), a size in points and a style, either #bold, #italic or #plain or a combination (addition) of them.

Examples:

```
font var0 <- font ('Helvetica Neue',12, #bold + #italic);   //
   var0 equals a bold and italic face of the Helvetica Neue
   family
```

---

frequency_of

Possible use:

- container frequency_of any expression —> map
- frequency_of (container , any expression) —> map

Result:

Returns a map with keys equal to the application of the right-hand argument (like collect) and values equal to the frequency of this key (i.e. how many times it has been obtained)

Examples:

```
map var0 <- [ag1, ag2, ag3, ag4] frequency_of each.size;    //
   var0 equals the different sizes as keys and the number of
   agents of this size as values
```

See also:

as_map,

---

fuzzy_kappa

Possible use:

- fuzzy_kappa (list<agent>, list, list, list<float>, list, matrix<float>, float) —> float
- fuzzy_kappa (list<agent>, list, list, list<float>, list, matrix<float>, float, list) —> float

Result:

fuzzy kappa indicator for 2 map comparisons: fuzzy_kappa(agents_list,list_vals1,list_-vals2, output_similarity_per_agents,categories,fuzzy_categories_matrix, fuzzy_distance, weights). Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21 fuzzy kappa indicator for 2 map comparisons: fuzzy_-kappa(agents_list,list_vals1,list_vals2, output_similarity_per_agents,categories,fuzzy_-categories_matrix, fuzzy_distance). Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21

Examples:

```
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2
   ],[cat2,cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,cat2
   ,cat3],[[1,0,0],[0,1,0],[0,0,1]], 2, [1.0,3.0,2.0,2.0,4.0])
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2
   ],[cat2,cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,cat2
   ,cat3],[[1,0,0],[0,1,0],[0,0,1]], 2)
```

fuzzy_kappa_sim

Possible use:

- fuzzy_kappa_sim (list<agent>, list, list, list, list<float>, list, matrix<float>, float) —> float
- fuzzy_kappa_sim (list<agent>, list, list, list, list<float>, list, matrix<float>, float, list) —> float

Result:

fuzzy kappa simulation indicator for 2 map comparisons: fuzzy_kappa_sim(agents_-list,list_vals1,list_vals2, output_similarity_per_agents,fuzzy_transitions_matrix, fuzzy_-distance, weights). Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800, fuzzy kappa simulation indicator for 2 map comparisons: fuzzy_kappa_sim(agents_list,list_vals1,list_vals2, output_similarity_per_agents,fuzzy_transitions_matrix, fuzzy_distance). Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800,

Examples:

```
fuzzy_kappa_sim([ag1, ag2, ag3, ag4, ag5], [cat1,cat1,cat2,cat3,
   cat2],[cat2,cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,
   cat2,cat3
   ],[[1,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0],[0,0,1,0,0,0,0,0,0],[0,0,0,1,0,0,0,0
    2,[1.0,3.0,2.0,2.0,4.0])
fuzzy_kappa_sim([ag1, ag2, ag3, ag4, ag5], [cat1,cat1,cat2,cat3,
   cat2],[cat2,cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,
   cat2,cat3
   ],[[1,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0],[0,0,1,0,0,0,0,0,0],[0,0,0,1,0,0,0,0
    2)
```

gaml_file

Possible use:

- gaml_file (string) —> file

Result:

Constructs a file of type gaml. Allowed extensions are limited to gaml

gamma_index

Possible use:

- gamma_index (graph) —> float

Result:

returns the gamma index of the graph (A measure of connectivity that considers the relationship between the number of observed links and the number of possible links: gamma = e/(3 ∗ (v - 2)) - for planar graph.

Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- gamma_index(graphEpidemio);   // var1 equals the
   gamma index of the graph
```

See also:

alpha_index, beta_index, nb_cycles, connectivity_index,

---

gauss

Possible use:

- gauss (point) —> float
- float gauss float —> float
- gauss (float , float) —> float

Result:

A value from a normally distributed random variable with expected value (mean) and variance (standardDeviation). The probability density function of such a variable is a Gaussian. A value from a normally distributed random variable with expected value (mean) and variance (standardDeviation). The probability density function of such a variable is a Gaussian.

Special cases:

- when the operand is a point, it is read as {mean, standardDeviation}
- when standardDeviation value is 0.0, it always returns the mean value

- when the operand is a point, it is read as {mean, standardDeviation}
- when standardDeviation value is 0.0, it always returns the mean value

Examples:

```
float var0 <- gauss(0,0.3);      // var0 equals 0.22354
float var1 <- gauss(0,0.3);      // var1 equals -0.1357
float var2 <- gauss({0,0.3});    // var2 equals 0.22354
float var3 <- gauss({0,0.3});    // var3 equals -0.1357
```

See also:

truncated_gauss, poisson,

`generate_barabasi_albert`

Possible use:

- `generate_barabasi_albert` (`container<agent>`, `species`, `int`, `bool`) —> `graph`
- `generate_barabasi_albert` (`species`, `species`, `int`, `int`, `bool`) —> `graph`

Result:

returns a random scale-free network (following Barabasi-Albert (BA) model). returns a random scale-free network (following Barabasi-Albert (BA) model).

Comment:

The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:

Special cases:

- "agents": list of existing node agents
- "edges_species": the species of edges
- "size": the graph will contain (size + 1) nodes
- "m": the number of edges added per novel node
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain (size + 1) nodes

- "m": the number of edges added per novel node
- "synchronized": is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<yourNodeSpecy,yourEdgeSpecy> graphEpidemio <-
   generate_barabasi_albert(
        yourListOfNodes,
        yourEdgeSpecy,
        3,
        5,
        true);
graph<yourNodeSpecy,yourEdgeSpecy> graphEpidemio <-
   generate_barabasi_albert(
        yourNodeSpecy,
        yourEdgeSpecy,
        3,
        5,
        true);
```

See also:

generate_watts_strogatz,

---

generate_complete_graph

Possible use:

- `generate_complete_graph` (`container<agent>`, `species`, `bool`) —> `graph`
- `generate_complete_graph` (`container<agent>`, `species`, `float`, `bool`) —> `graph`
- `generate_complete_graph` (`species`, `species`, `int`, `bool`) —> `graph`
- `generate_complete_graph` (`species`, `species`, `int`, `float`, `bool`) —> `graph`

Result:

returns a fully connected graph. returns a fully connected graph. returns a fully connected graph. returns a fully connected graph.

Comment:

Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:

Special cases:

- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain size nodes.
- "layoutRadius": nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "agents": list of existing node agents
- "edges_species": the species of edges
- "layoutRadius": nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "vertices_specy": the species of vertices
- "edges_species": the species of edges

- "size": the graph will contain size nodes.
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "agents": list of existing node agents
- "edges_species": the species of edges
- "synchronized": is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_complete_graph(
            myVertexSpecy,
            myEdgeSpecy,
            10, 25,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_complete_graph(
            myListOfNodes,
            myEdgeSpecy,
            25,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_complete_graph(
            myVertexSpecy,
            myEdgeSpecy,
            10,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_complete_graph(
            myListOfNodes,
            myEdgeSpecy,
        true);
```

See also:

generate_barabasi_albert, generate_watts_strogatz,

`generate_watts_strogatz`

Possible use:

- `generate_watts_strogatz` (`container<agent>`, `species`, `float`, `int`, `bool`) —> `graph`
- `generate_watts_strogatz` (`species`, `species`, `int`, `float`, `int`, `bool`) —> `graph`

Result:

returns a random small-world network (following Watts-Strogatz model). returns a random small-world network (following Watts-Strogatz model).

Comment:

The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:

Special cases:

- "agents": list of existing node agents

- "edges_species": the species of edges
- "p": probability to "rewire" an edge. So it must be between 0 and 1. The parameter is often called beta in the literature.
- "k": the base degree of each node. k must be greater than 2 and even.
- "synchronized": is the graph and the species of vertices and edges synchronized?
- "vertices_specy": the species of vertices
- "edges_species": the species of edges
- "size": the graph will contain (size + 1) nodes. Size must be greater than k.
- "p": probability to "rewire" an edge. So it must be between 0 and 1. The parameter is often called beta in the literature.
- "k": the base degree of each node. k must be greater than 2 and even.
- "synchronized": is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_watts_strogatz(
            myListOfNodes,
            myEdgeSpecy,
            0.3,
            2,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <-
   generate_watts_strogatz(
            myVertexSpecy,
            myEdgeSpecy,
            2,
            0.3,
            2,
        true);
```

See also:

generate_barabasi_albert,

---

geometric_mean

Possible use:

- geometric_mean (container) —> float

Result:

the geometric mean of the elements of the operand. See Geometric_mean for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- geometric_mean ([4.5, 3.5, 5.5, 7.0]);    // var0
   equals 4.962326343467649
```

See also:

mean, median, harmonic_mean,

---

geometry

Possible use:

- geometry (any) —> geometry

Result:

Casts the operand into the type geometry

---

geometry_collection

Possible use:

- geometry_collection (container<geometry>) —> geometry

Result:

A geometry collection (multi-geometry) composed of the given list of geometries.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single geometry, returns a copy of the geometry.

Examples:

```
geometry var0 <- geometry_collection([{0,0}, {0,10}, {10,10},
   {10,0}]);      // var0 equals a geometry composed of the 4
   points (multi-point).
```

See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle, line,

---

get

Possible use:

- agent get string —> unknown
- get (agent , string) —> unknown
- geometry get string —> unknown
- get (geometry , string) —> unknown

Result:

Reads an attribute of the specified agent (left operand). The attribute name is specified by the right operand. Reads an attribute of the specified geometry (left operand). The attribute name is specified by the right operand.

Special cases:

- Reading the attribute of another agent

```
string agent_name <- an_agent get('name');     // reads then '
   name' attribute of an_agent then assigns the returned value to
    the agent_name variable
```

- Reading the attribute of a geometry

```
string geom_area <- a_geometry get('area');     // reads then '
   area' attribute of 'a_geometry' variable then assigns the
   returned value to the geom_area variable
```

---

get_about

Possible use:

- get_about (emotion) —> predicate

Result:

get the about value of the given emotion

Examples:

```
emotion set_intensity 12
```

---

get_decay

Possible use:

- get_decay (emotion) —> float

Result:

get the decay value of the given emotion

Examples:

```
emotion set_intensity 12
```

`get_intensity`

Possible use:

- `get_intensity` (`emotion`) —> `float`

Result:

get the intensity value of the given emotion

Examples:

```
emotion set_intensity 12
```

`get_lifetime`

Possible use:

- `get_lifetime` (`predicate`) —> `int`

`get_priority`

Possible use:

- `get_priority` (`predicate`) —> `float`

`get_super_intention`

Possible use:

- `get_super_intention` (`predicate`) —> `predicate`

`graph`

Possible use:

- `graph` (`any`) —> `graph`

Result:

Casts the operand into the type graph

`grayscale`

Possible use:

- `grayscale` (`rgb`) —> `rgb`

Result:

Converts rgb color to grayscale value

Comment:

r=red, g=green, b=blue. Between 0 and 255 and gray = 0.299 ∗ red + 0.587 ∗ green + 0.114 ∗ blue (Photoshop value)

Examples:

```
rgb var0 <- grayscale (rgb(255,0,0));    // var0 equals to a dark
   grey
```

See also:

rgb, hsb,

---

grid_at

Possible use:

- species grid_at point —> agent
- grid_at (species , point) —> agent

Result:

returns the cell of the grid (right-hand operand) at the position given by the right-hand operand

Comment:

If the left-hand operand is a point of floats, it is used as a point of ints.

Special cases:

- if the left-hand operand is not a grid cell species, returns nil

Examples:

```
agent var0 <- grid_cell grid_at {1,2};  // var0 equals the agent
   grid_cell with grid_x=1 and grid_y = 2
```

---

grid_cells_to_graph

Possible use:

- grid_cells_to_graph (container) —> graph

Result:

creates a graph from a list of cells (operand). An edge is created between neighbors.

Examples:

```
my_cell_graph<-grid_cells_to_graph(cells_list)
```

---

grid_file

Possible use:

- grid_file (string) —> file

Result:

Constructs a file of type grid. Allowed extensions are limited to asc, tif

---

group_by

Possible use:

- container group_by any expression —> map
- group_by (container , any expression) —> map

Result:

Returns a map, where the keys take the possible values of the right-hand operand and the map values are the list of elements of the left-hand operand associated to the key value

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, group_by throws an error

Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] group_by (each > 3);  // var0
   equals [false::[1, 2, 3], true::[4, 5, 6, 7, 8]]
map var1 <- g2 group_by (length(g2 out_edges_of each) );    //
   var1 equals [ 0::[node9, node7, node10, node8, node11], 1::[
   node6], 2::[node5], 3::[node4]]
map var2 <- (list(node) group_by (round(node(each).location.x));
      // var2 equals [32::[node5], 21::[node1], 4::[node0], 66::[
   node2], 96::[node3]]
map var3 <- [1::2, 3::4, 5::6] group_by (each > 4);     // var3
   equals [false::[2, 4], true::[6]]
```

See also:

first_with, last_with, where,

---

harmonic_mean

Possible use:

- `harmonic_mean` (`container`) —> `float`

Result:

the harmonic mean of the elements of the operand. See Harmonic_mean for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- harmonic_mean ([4.5, 3.5, 5.5, 7.0]);     // var0
   equals 4.804159445407279
```

See also:

mean, median, geometric_mean,

---

hexagon

Possible use:

- `hexagon (point) —> geometry`
- `hexagon (float) —> geometry`

Result:

A hexagon geometry which the given with and height

Comment:

the center of the hexagon is by default the location of the current agent in which has been called this operator.the center of the hexagon is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- hexagon({10,5});   // var0 equals a geometry as
   a hexagon of width of 10 and height of 5.
geometry var1 <- hexagon(10);   // var1 equals a geometry as a
   hexagon of width of 10 and height of 10.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

hierarchical_clustering

Possible use:

- container<agent> hierarchical_clustering float —> container
- hierarchical_clustering (container<agent> , float) —> container

Result:

A tree (list of list) contained groups of agents clustered by distance considering a distance min between two groups.

Comment:

use of hierarchical clustering with Minimum for linkage criterion between two groups of agents.

Examples:

```
container var0 <- [ag1, ag2, ag3, ag4, ag5]
   hierarchical_clustering 20.0;   // var0 equals for example,
   can return [[[ag1],[ag3]], [ag2], [[[ag4],[ag5]],[ag6]]
```

See also:

simple_clustering_by_distance,

---

hsb

Possible use:

- hsb (float, float, float) —> rgb
- hsb (float, float, float, int) —> rgb
- hsb (float, float, float, float) —> rgb

Result:

Converts hsb (h=hue, s=saturation, b=brightness) value to Gama color

Comment:

h,s and b components should be floating-point values between 0.0 and 1.0 and when used alpha should be an integer (between 0 and 255) or a float (between 0 and 1) . Examples: Red=(0.0,1.0,1.0), Yellow=(0.16,1.0,1.0), Green=(0.33,1.0,1.0), Cyan=(0.5,1.0,1.0), Blue=(0.66,1.0,1.0), Magenta=(0.83,1.0,1.0)

Examples:

```
rgb var0 <- hsb (0.0,1.0,1.0);  // var0 equals rgb("red")
rgb var1 <- hsb (0.5,1.0,1.0,0.0);  // var1 equals rgb("cyan",0)
```

See also:

rgb,

`hypot`

Possible use:

- `hypot` (`float`, `float`, `float`, `float`) —> `float`

Result:

Returns sqrt(x2 +y2) without intermediate overflow or underflow.

Special cases:

- If either argument is infinite, then the result is positive infinity. If either argument is NaN and neither argument is infinite, then the result is NaN.

Examples:

```
float var0 <- hypot(0,1,0,1);   // var0 equals sqrt(2)
```

IDW

Possible use:

- `IDW` (`container<agent>`, `map<point,float>`, `int`) —> `map<agent,float>`

Result:

Inverse Distance Weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to each geometry are calculated with a weighted average of the values available at the known points. See: http://en.wikipedia.org/wiki/Inverse_distance_weighting Usage: IDW (list of geometries, map of points (key: point, value: value), power parameter)

Examples:

```
map<agent,float> var0 <- IDW([ag1, ag2, ag3, ag4, ag5
   ],[{10,10}::25.0, {10,80}::10.0, {100,10}::15.0], 2);  // var0
    equals for example, can return [ag1::12.0, ag2::23.0,ag3
   ::12.0,ag4::14.0,ag5::17.0]
```

---

image_file

Possible use:

- `image_file` (`string`) —> `file`

Result:

Constructs a file of type image. Allowed extensions are limited to tiff, jpg, jpeg, png, gif, pict, bmp

---

`in`

Possible use:

- `string` `in` `string` —> `bool`
- `in` (`string` , `string`) —> `bool`
- `unknown` `in` `container` —> `bool`
- `in` (`unknown` , `container`) —> `bool`

Result:

true if the right operand contains the left operand, false otherwise

Comment:

the definition of in depends on the container

Special cases:

- if both operands are strings, returns true if the left-hand operand patterns is included in to the right-hand string;

- if the right operand is nil or empty, in returns false

Examples:

```
bool var0 <-  'bc' in 'abcded';      // var0 equals true
bool var1 <- 2 in [1,2,3,4,5,6];     // var1 equals true
bool var2 <- 7 in [1,2,3,4,5,6];     // var2 equals false
bool var3 <- 3 in [1::2, 3::4, 5::6];    // var3 equals false
bool var4 <- 6 in [1::2, 3::4, 5::6];    // var4 equals true
```

See also:

contains,

---

in_degree_of

Possible use:

- graph in_degree_of unknown —> int
- in_degree_of (graph , unknown) —> int

Result:

returns the in degree of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap in_degree_of (node(3));     // var1
   equals 2
```

See also:

out_degree_of, degree_of,

---

in_edges_of

Possible use:

- graph in_edges_of unknown —> container
- in_edges_of (graph , unknown) —> container

Result:

returns the list of the in-edges of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
container var1 <- graphFromMap in_edges_of node({12,45});   //
   var1 equals [LineString]
```

See also:

out_edges_of,

---

indented_by

Possible use:

- string indented_by int —> string
- indented_by (string , int) —> string

Result:

Converts a (possibly multiline) string by indenting it by a number – specified by the second operand – of tabulations to the right

---

`index_by`

Possible use:

- `container index_by any expression —> map`
- `index_by (container , any expression) —> map`

Result:

produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand

Special cases:

- if the left-hand operand is nil, index_by throws an error.

Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] index_by (each - 1);  // var0
   equals [0::1, 1::2, 2::3, 3::4, 4::5, 5::6, 6::7, 7::8]
```

`index_of`

Possible use:

- `string index_of string —> int`
- `index_of (string , string) —> int`
- `container index_of unknown —> int`
- `index_of (container , unknown) —> int`
- `matrix index_of unknown —> point`
- `index_of (matrix , unknown) —> point`
- `species index_of unknown —> int`
- `index_of (species , unknown) —> int`
- `map index_of unknown —> unknown`
- `index_of (map , unknown) —> unknown`

Result:

the index of the first occurence of the right operand in the left operand container the index of the first occurence of the right operand in the left operand container

Comment:

The definition of index_of and the type of the index depend on the container

Special cases:

- if the left operator is a species, returns the index of an agent in a species. If the argument is not an agent of this species, returns -1. Use int(agent) instead
- if the left operand is a map, index_of returns the index of a value or nil if the value is not mapped
- if both operands are strings, returns the index within the left-hand string of the first occurrence of the given right-hand string

```
int var0 <-  "abcabcabc" index_of "ca";     // var0 equals 2
```

- if the left operand is a list, index_of returns the index as an integer

```
int var1 <- [1,2,3,4,5,6] index_of 4;   // var1 equals 3
int var2 <- [4,2,3,4,5,4] index_of 4;   // var2 equals 0
```

- if the left operand is a matrix, index_of returns the index as a point

```
point var3 <- matrix([[1,2,3],[4,5,6]]) index_of 4;     // var3
   equals {1.0,0.0}
```

Examples:

```
unknown var4 <- [1::2, 3::4, 5::6] index_of 4;  // var4 equals 3
```

See also:

at, last_index_of,

`inside`

Possible use:

- `container<agent> inside geometry —> list<geometry>`
- `inside (container<agent> , geometry) —> list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), covered by the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] inside(self);    // var0
   equals the agents among ag1, ag2 and ag3 that are covered by
   the shape of the right-hand argument.
list<geometry> var1 <- (species1 + species2) inside (self);
   // var1 equals the agents among species species1 and species2
   that are covered by the shape of the right-hand argument.
```

See also:

neighbors_at, neighbors_of, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

`int`

Possible use:

- `int` (`any`) —> `int`

Result:

Casts the operand into the type int

`inter`

Possible use:

- `geometry` `inter` `geometry` —> `geometry`
- `inter` (`geometry` , `geometry`) —> `geometry`
- `container` `inter` `container` —> `container`
- `inter` (`container` , `container`) —> `container`

Result:

A geometry resulting from the intersection between the two geometries the intersection of the two operands

Comment:

both containers are transformed into sets (so without duplicated element, cf. remove_deplicates operator) before the set intersection is computed.

Special cases:

- returns nil if one of the operands is nil

- if an operand is a graph, it will be transformed into the set of its nodes
- if an operand is a map, it will be transformed into the set of its values

```
container var3 <- [1::2, 3::4, 5::6] inter [2,4];   // var3
   equals [2,4]
container var4 <- [1::2, 3::4, 5::6] inter [1,3];   // var4
   equals []
```

- if an operand is a matrix, it will be transformed into the set of the lines

```
container var5 <- matrix([[1,2,3],[4,5,4]]) inter [3,4];    //
   var5 equals [3,4]
```

Examples:

```
geometry var0 <- square(10) inter circle(5);    // var0 equals
   circle(5)
container var1 <- [1,2,3,4,5,6] inter [2,4];    // var1 equals
   [2,4]
container var2 <- [1,2,3,4,5,6] inter [0,8];    // var2 equals []
```

See also:

union, +, -, remove_duplicates,

`interleave`

Possible use:

- `interleave` (`container`) —> `container`

Result:

a new list containing the interleaved elements of the containers contained in the operand

Comment:

the operand should be a list of lists of elements. The result is a list of elements.

Examples:

```
container var0 <- interleave([1,2,4,3,5,7,6,8]);     // var0
   equals [1,2,4,3,5,7,6,8]
container var1 <- interleave([['e11','e12','e13'],['e21','e22','
   e23'],['e31','e32','e33']]);     // var1 equals ['e11','e21','
   e31','e12','e22','e32','e13','e23','e33']
```

`internal_at`

Possible use:

- `agent` `internal_at` `container` —> `unknown`
- `internal_at` (`agent` , `container`) —> `unknown`
- `container<KeyType,ValueType>` `internal_at` `list<KeyType>` —> `ValueType`
- `internal_at` (`container<KeyType,ValueType>` , `list<KeyType>`) —> `ValueType`
- `geometry` `internal_at` `container` —> `unknown`
- `internal_at` (`geometry` , `container`) —> `unknown`

Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with index For internal use only. Corresponds to the implementation of the access to containers with index For internal use only. Corresponds to the implementation, for geometries, of the access to containers with index

See also:

at,

`internal_integrated_value`

Possible use:

- `any expression` `internal_integrated_value` `any expression` —> `container`
- `internal_integrated_value` (`any expression` , `any expression`) —> `container`

Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with index

---

`internal_zero_order_equation`

Possible use:

- `internal_zero_order_equation` (`any expression`) —> `float`

`intersection`

Same signification as inter

`intersects`

Possible use:

- `geometry` `intersects` `geometry` —> `bool`
- `intersects` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) intersects the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- square(5) intersects {10,10};  // var0 equals false
```

See also:

disjoint_from, crosses, overlaps, partially_overlaps, touches,

---

`inverse`

Possible use:

- `inverse` (`matrix`) —> `matrix<float>`

Result:

The inverse matrix of the given matrix. If no inverse exists, returns a matrix that has properties that resemble that of an inverse.

Examples:

```
matrix<float> var0 <- inverse(matrix([[5,-3],[6,-4]]));     //
   var0 equals [2.0000000000000004,-0.9999999999999998]
```

---

inverse_distance_weighting

Same signification as IDW

---

is

Possible use:

- unknown is any expression —> bool
- is (unknown , any expression) —> bool

Result:

returns true if the left operand is of the right operand type, false otherwise

Examples:

```
bool var0 <- 0 is int;  // var0 equals true
bool var1 <- an_agent is node;  // var1 equals true
bool var2 <- 1 is float;    // var2 equals false
```

---

is_clockwise

Possible use:

- `is_clockwise` (`geometry`) —> `bool`

Result:

returns true if the geometry is defined clockwise

Examples:

```
bool var0 <- is_clockwise(circle(10)); // var0 equals true
```

See also:

change_clockwise,

---

is_csv

Possible use:

- `is_csv` (`any`) —> `bool`

Result:

Tests whether the operand is a csv file.

---

is_dxf

Possible use:

- is_dxf (any) —> bool

Result:

Tests whether the operand is a dxf file.

---

is_finite

Possible use:

- is_finite (float) —> bool

Result:

Returns whether the argument is a finite number or not

Examples:

```
bool var0 <- is_finite(4.66);   // var0 equals true
bool var1 <- is_finite(#infinity);  // var1 equals false
```

---

is_gaml

Possible use:

- is_gaml (any) —> bool

Result:

Tests whether the operand is a gaml file.

`is_grid`

Possible use:

- `is_grid` (`any`) —> `bool`

Result:

Tests whether the operand is a grid file.

`is_image`

Possible use:

- `is_image` (`any`) —> `bool`

Result:

Tests whether the operand is a image file.

`is_number`

Possible use:

- `is_number` (`float`) —> `bool`
- `is_number` (`string`) —> `bool`

Result:

Returns whether the argument is a real number or not tests whether the operand represents a numerical value

Comment:

Note that the symbol . should be used for a float value (a string with , will not be considered as a numeric value). Symbols e and E are also accepted. A hexadecimal value should begin with #.

Examples:

```
bool var0 <- is_number(4.66);   // var0 equals true
bool var1 <- is_number(#infinity);  // var1 equals true
bool var2 <- is_number(#nan);   // var2 equals false
bool var3 <- is_number("test");     // var3 equals false
bool var4 <- is_number("123.56");   // var4 equals true
bool var5 <- is_number("-1.2e5");   // var5 equals true
bool var6 <- is_number("1,2");  // var6 equals false
bool var7 <- is_number("#12FA");    // var7 equals true
```

`is_obj`

Possible use:

- `is_obj` (`any`) —> `bool`

Result:

Tests whether the operand is a obj file.

is_osm

Possible use:

- `is_osm` (`any`) —> `bool`

Result:

Tests whether the operand is a osm file.

---

is_pgm

Possible use:

- `is_pgm` (`any`) —> `bool`

Result:

Tests whether the operand is a pgm file.

---

is_property

Possible use:

- `is_property` (`any`) —> `bool`

Result:

Tests whether the operand is a property file.

`is_R`

Possible use:

- `is_R` (`any`) —> `bool`

Result:

Tests whether the operand is a R file.

`is_shape`

Possible use:

- `is_shape` (`any`) —> `bool`

Result:

Tests whether the operand is a shape file.

`is_skill`

Possible use:

- `unknown` `is_skill` `string` —> `bool`
- `is_skill` (`unknown` , `string`) —> `bool`

Result:

returns true if the left operand is an agent whose species implements the right-hand skill name

Examples:

```
bool var0 <- agentA is_skill 'moving';  // var0 equals true
```

---

is_svg

Possible use:

- `is_svg` (`any`) —> `bool`

Result:

Tests whether the operand is a svg file.

---

is_text

Possible use:

- `is_text` (`any`) —> `bool`

Result:

Tests whether the operand is a text file.

---

is_threeds

Possible use:

- `is_threeds` (`any`) —> `bool`

Result:

Tests whether the operand is a threeds file.

---

is_URL

Possible use:

- `is_URL` (`any`) —> `bool`

Result:

Tests whether the operand is a URL file.

---

is_xml

Possible use:

- `is_xml` (`any`) —> `bool`

Result:

Tests whether the operand is a xml file.

---

kappa

Possible use:

- `kappa` (`list`, `list`, `list`) —> `float`
- `kappa` (`list`, `list`, `list`, `list`) —> `float`

Result:

kappa indicator for 2 map comparisons: kappa(list_vals1,list_vals2,categories, weights). Reference: Cohen, J. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 1960, 20. kappa indicator for 2 map comparisons: kappa(list_vals1,list_vals2,categories). Reference: Cohen, J. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 1960, 20.

Examples:

```
kappa([cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],[cat1
   ,cat2,cat3], [1.0, 2.0, 3.0, 1.0, 5.0])
kappa([cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],[cat1
   ,cat2,cat3])
float var2 <- kappa([1,3,5,1,5],[1,1,1,1,5],[1,3,5]);   // var2
   equals the similarity between 0 and 1
float var3 <- kappa([1,1,1,1,5],[1,1,1,1,5],[1,3,5]);   // var3
   equals 1.0
```

---

kappa_sim

Possible use:

- `kappa_sim` (`list`, `list`, `list`, `list`) —> `float`
- `kappa_sim` (`list`, `list`, `list`, `list`, `list`) —> `float`

Result:

kappa simulation indicator for 2 map comparisons: kappa(list_valsInits,list_valsObs,list_-valsSim, categories). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8). kappa simulation indicator for 2 map comparisons: kappa(list_valsInits,list_valsObs,list_valsSim, categories, weights). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8)

Examples:

```
kappa([cat1,cat1,cat2,cat2,cat2],[cat2,cat1,cat2,cat1,cat3],[cat2
   ,cat1,cat2,cat3,cat3], [cat1,cat2,cat3])
kappa([cat1,cat1,cat2,cat2,cat2],[cat2,cat1,cat2,cat1,cat3],[cat2
   ,cat1,cat2,cat3,cat3], [cat1,cat2,cat3],[1.0, 2.0, 3.0, 1.0,
   5.0])
```

`kmeans`

Possible use:

- `list` `kmeans` `int` —> `list<list>`
- `kmeans` (`list` , `int`) —> `list<list>`
- `kmeans` (`list`, `int`, `int`) —> `list<list>`

Result:

returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k) and the maximum number of iterations to run the algorithm for (If negative, no maximum will be used) (maxIt). Usage: kmeans(data,k,maxit) returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k). Usage: kmeans(data,k)

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0
- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2,10)
kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2)
```

---

kurtosis

Possible use:

- `kurtosis` (`list`) —> `float`

Result:

returns kurtosis value computed from the operand list of values

Special cases:

- if the length of the list is lower than 3, returns NaN

Examples:

```
kurtosis ([1,2,3,4,5])
```

---

last

Possible use:

- `last` (`string`) —> `string`
- `last` (`container<KeyType,ValueType>`) —> `ValueType`
- `int last container` —> `container`
- `last` (`int` , `container`) —> `container`

Result:

the last element of the operand

Comment:

the last operator behavior depends on the nature of the operand

Special cases:

- if it is a map, last returns the value of the last pair (in insertion order)
- if it is a file, last returns the last element of the content of the file (that is also a container)
- if it is a population, last returns the last agent of the population
- if it is a graph, last returns a list containing the last edge created
- if it is a matrix, last returns the element at {length-1,length-1} in the matrix
- for a matrix of int or float, it will return 0 if the matrix is empty
- for a matrix of object or geometry, it will return nil if the matrix is empty
- if it is a string, last returns a string composed of its last character, or an empty string if the operand is empty

```
string var0 <- last ('abce');   // var0 equals 'e'
```

- if it is a list, last returns the last element of the list, or nil if the list is empty

```
int var1 <- last ([1, 2, 3]);   // var1 equals 3
```

See also:

first,

`last_index_of`

Possible use:

- `string` `last_index_of` `string` —> `int`
- `last_index_of` (`string` , `string`) —> `int`
- `matrix` `last_index_of` `unknown` —> `point`
- `last_index_of` (`matrix` , `unknown`) —> `point`
- `container` `last_index_of` `unknown` —> `int`
- `last_index_of` (`container` , `unknown`) —> `int`
- `map` `last_index_of` `unknown` —> `unknown`
- `last_index_of` (`map` , `unknown`) —> `unknown`
- `species` `last_index_of` `unknown` —> `int`
- `last_index_of` (`species` , `unknown`) —> `int`

Result:

the index of the last occurence of the right operand in the left operand container

Comment:

The definition of last_index_of and the type of the index depend on the container

Special cases:

- if the left operand is a species, the last index of an agent is the same as its index
- if both operands are strings, returns the index within the left-hand string of the rightmost occurrence of the given right-hand string

```
int var0 <- "abcabcabc" last_index_of "ca";      // var0 equals 5
```

- if the left operand is a matrix, last_index_of returns the index as a point

```
point var1 <- matrix([[1,2,3],[4,5,4]]) last_index_of 4;     //
   var1 equals {1.0,2.0}
```

- if the left operand is a list, last_index_of returns the index as an integer

```
int var2 <- [1,2,3,4,5,6] last_index_of 4;  // var2 equals 3
int var3 <- [4,2,3,4,5,4] last_index_of 4;  // var3 equals 5
```

- if the left operand is a map, last_index_of returns the index as an int (the key of the pair)

```
unknown var4 <- [1::2, 3::4, 5::4] last_index_of 4;     // var4
   equals 5
```

See also:

at, last_index_of, index_of,

last_with

Possible use:

- `container` `last_with` `any expression` —> `unknown`
- `last_with` (`container` , `any expression`) —> `unknown`

Result:

the last element of the left-hand operand that makes the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, last_with throws an error.
- If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] last_with (each >= 4);   //
   var4 equals 6
unknown var5 <- [1::2, 3::4, 5::6].pairs last_with (each.value >=
    4);   // var5 equals 5::6
```

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] last_with (each > 3);     //
   var0 equals 8
unknown var2 <- g2 last_with (length(g2 out_edges_of each) = 0 );
      // var2 equals node11
unknown var3 <- (list(node) last_with (round(node(each).location.
   x) > 32);  // var3 equals node3
```

See also:

group_by, first_with, where,

`layout`

Possible use:

- `graph` `layout` `string` —> `graph`
- `layout` (`graph` , `string`) —> `graph`
- `layout` (`graph`, `string`, `int`) —> `graph`
- `layout` (`graph`, `string`, `int`, `map<string,unknown>`) —> `graph`

Result:

layouts a GAMA graph.

`length`

Possible use:

- `length` (`container<KeyType,ValueType>`) —> `int`
- `length` (`string`) —> `int`

Result:

the number of elements contained in the operand

Comment:

the length operator behavior depends on the nature of the operand

Special cases:

- if it is a population, length returns number of agents of the population

- if it is a graph, length returns the number of vertexes or of edges (depending on the way it was created)
- if it is a list or a map, length returns the number of elements in the list or map

```
int var0 <- length([12,13]);     // var0 equals 2
int var1 <- length([]);     // var1 equals 0
```

- if it is a matrix, length returns the number of cells

```
int var2 <- length(matrix([["c11","c12","c13"],["c21","c22","c23"
   ]]));  // var2 equals 6
```

- if it is a string, length returns the number of characters

```
int var3 <- length ('I am an agent');   // var3 equals 13
```

`line`

Possible use:

- `line` (`container<geometry>`) —> `geometry`
- `container<geometry>` `line` `float` —> `geometry`
- `line` (`container<geometry>` , `float`) —> `geometry`

Result:

A polyline geometry from the given list of points represented as a cylinder of radius r. A polyline geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.
- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.
- if a radius is added, the given list of points represented as a cylinder of radius r

```
geometry var0 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}],0.2);
        // var0 equals a polyline geometry composed of the 4 points
    .
```

Examples:

```
geometry var1 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}]);
    // var1 equals a polyline geometry composed of the 4 points.
```

See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle,

---

`link`

Possible use:

- `geometry` **`link`** `geometry` —> `geometry`
- **`link`** (`geometry` , `geometry`) —> `geometry`

Result:

A dynamic line geometry between the location of the two operands

Comment:

The geometry of the link is a line between the locations of the two operands, which is built and maintained dynamically

Special cases:

- if one of the operands is nil, link returns a point geometry at the location of the other. If both are null, it returns a point geometry at {0,0}

Examples:

```
geometry var0 <- link (geom1,geom2);    // var0 equals a link
   geometry between geom1 and geom2.
```

See also:

around, circle, cone, line, norm, point, polygon, polyline, rectangle, square, triangle,

---

list

Possible use:

- `list` (any) —> `list`

Result:

Casts the operand into the type list

`list_with`

Possible use:

- `int list_with any expression —> container`
- `list_with (int , any expression) —> container`

Result:

creates a list with a size provided by the first operand, and filled with the second operand

Comment:

Note that the right operand should be positive, and that the second one is evaluated for each position in the list.

See also:

list,

`ln`

Possible use:

- `ln (float) —> float`
- `ln (int) —> float`

Result:

Returns the natural logarithm (base e) of the operand.

Special cases:

- an exception is raised if the operand is less than zero.

Examples:

```
float var0 <- ln(exp(1));   // var0 equals 1.0
float var1 <- ln(1);    // var1 equals 0.0
```

See also:

exp,

`load_graph_from_file`

Possible use:

- `load_graph_from_file` (`string`) —> `graph`
- `string` `load_graph_from_file` `file` —> `graph`
- `load_graph_from_file` (`string` , `file`) —> `graph`
- `string` `load_graph_from_file` `string` —> `graph`
- `load_graph_from_file` (`string` , `string`) —> `graph`
- `load_graph_from_file` (`string`, `species`, `species`) —> `graph`
- `load_graph_from_file` (`string`, `file`, `species`, `species`) —> `graph`
- `load_graph_from_file` (`string`, `string`, `species`, `species`) —> `graph`
- `load_graph_from_file` (`string`, `string`, `species`, `species`, `bool`) —> `graph`

Result:

returns a graph loaded from a given file encoded into a given format. The last boolean parameter indicates whether the resulting graph will be considered as spatial or not by GAMA loads a graph from a file

Comment:

Available formats: "pajek": Pajek (Slovene word for Spider) is a program, for Windows, for analysis and visualization of large networks. See: http://pajek.imfm.si/doku.php?id=pajek for more details."lgl": LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details."dot": DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. See: http://en.wikipedia.org/wiki/DOT_-language for more details."edge": This format is a simple text file with numeric vertex ids defining the edges."gexf": GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications. See: http://gexf.net/format/ for more details."graphml": GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details."tlp" or "tulip": TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. "ncol": This format is used by the Large Graph Layout progra. It is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge. See: http://bioinformatics.icmb.utexas.edu/lgl for more details.The map operand should includes following elements:Available formats: "pajek": Pajek (Slovene word for Spider) is a program, for Windows, for analysis and visualization of large networks. See: http://pajek.imfm.si/doku.php?id=pajek for more details."lgl": LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details."dot": DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. See: http://en.wikipedia.org/wiki/DOT_-language for more details."edge": This format is a simple text file with numeric vertex ids defining the edges."gexf": GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications. See: http://gexf.net/format/ for more details."graphml": GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details."tlp" or "tulip": TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. "ncol": This format is used by the Large Graph Layout progra. It is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge. See: http://bioinformatics.icmb.utexas.edu/lgl for more details.The map operand should includes following elements:

Special cases:

- "format": the format of the file
- "filename": the filename of the file containing the network
- "edges_species": the species of edges
- "vertices_specy": the species of vertices
- "format": the format of the file
- "filename": the filename of the file containing the network
- "edges_species": the species of edges
- "vertices_specy": the species of vertices
- "filename": the filename of the file containing the network, "edges_species": the species of edges, "vertices_specy": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy );
```

- "format": the format of the file, "file": the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

- “format”: the format of the file, “file”: the file containing the network, “edges_species”: the species of edges, “vertices_specy”: the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy );
```

- “file”: the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

- “format”: the format of the file, “filename”: the filename of the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy , true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy);
```

`load_shortest_paths`

Possible use:

- `graph` `load_shortest_paths` `matrix` —> `graph`
- `load_shortest_paths` (`graph` , `matrix`) —> `graph`

Result:

put in the graph cache the computed shortest paths contained in the matrix (rows: source, columns: target)

Examples:

```
graph var0 <- load_shortest_paths(shortest_paths_matrix);   //
    var0 equals return my_graph with all the shortest paths
    computed
```

`log`

Possible use:

- `log` (`float`) —> `float`
- `log` (`int`) —> `float`

Result:

Returns the logarithm (base 10) of the operand.

Special cases:

- an exception is raised if the operand is equals or less than zero.

Examples:

```
float var0 <- log(10);   // var0 equals 1.0
float var1 <- log(1);    // var1 equals 0.0
```

See also:

ln,

`lower_case`

Possible use:

- `lower_case` (`string`) —> `string`

Result:

Converts all of the characters in the string operand to lower case

Examples:

```
string var0 <- lower_case("Abc");    // var0 equals 'abc'
```

See also:

upper_case,

---

map

Possible use:

- map (any) —> map

Result:

Casts the operand into the type map

---

masked_by

Possible use:

- geometry masked_by container<geometry> —> geometry
- masked_by (geometry , container<geometry>) —> geometry
- masked_by (geometry, container<geometry>, int) —> geometry

Examples:

```
geometry var0 <- perception_geom masked_by obstacle_list;   //
   var0 equals the geometry representing the part of
   perception_geom visible from the agent position considering
   the list of obstacles obstacle_list.
geometry var1 <- perception_geom masked_by obstacle_list;   //
   var1 equals the geometry representing the part of
   perception_geom visible from the agent position considering
   the list of obstacles obstacle_list.
```

`material`

Possible use:

- `float` **`material`** `float` —> `msi.gama.util.GamaMaterial`
- **`material`** (`float` , `float`) —> `msi.gama.util.GamaMaterial`

Result:

Returns

Examples:

```
```

See also:

,

`matrix`

Possible use:

- **`matrix`** (`any`) —> `matrix`

Result:

Casts the operand into the type matrix

`matrix_with`

Possible use:

- `point` `matrix_with` `any expression` —> `matrix`
- `matrix_with` (`point` , `any expression`) —> `matrix`

Result:

creates a matrix with a size provided by the first operand, and filled with the second operand

Comment:

Note that both components of the right operand point should be positive, otherwise an exception is raised.

See also:

matrix, as_matrix,

`max`

Possible use:

- `max` (`container`) —> `unknown`

Result:

the maximum element found in the operand

Comment:

the max operator behavior depends on the nature of the operand

Special cases:

- if it is a population of a list of other type: max transforms all elements into integer and returns the maximum of them
- if it is a map, max returns the maximum among the list of all elements value
- if it is a file, max returns the maximum of the content of the file (that is also a container)
- if it is a graph, max returns the maximum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, max returns the maximum of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, max returns the maximum of the list of the geometries
- if it is a matrix of another type, max returns the maximum of the elements transformed into float
- if it is a list of int of float, max returns the maximum of all the elements

```
unknown var0 <- max ([100, 23.2, 34.5]);    // var0 equals 100.0
```

- if it is a list of points: max returns the maximum of all points as a point (i.e. the point with the greatest coordinate on the x-axis, in case of equality the point with the greatest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)

```
unknown var1 <- max([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]);
      // var1 equals {9.0,1.0}
```

See also:

min,

max_of

Possible use:

- container max_of any expression —> unknown
- max_of (container , any expression) —> unknown

Result:

the maximum value of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- As of GAMA 1.6, if the left-hand operand is nil or empty, max_of throws an error
- if the left-operand is a map, the keyword each will contain each value

```
unknown var5 <- [1::2, 3::4, 5::6] max_of (each + 3);   // var5
    equals 6
```

Examples:

```
unknown var1 <- [1,2,4,3,5,7,6,8] max_of (each * 100 );     //
    var1 equals 800
graph g2 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
unknown var3 <- g2.vertices max_of (g2 degree_of( each ));  //
    var3 equals 2
unknown var4 <- (list(node) max_of (round(node(each).location.x))
    ;  // var4 equals 96
```

See also:

min_of,

---

maximal_cliques_of

Possible use:

- `maximal_cliques_of` (`graph`) —> `list<list>`

Result:

returns the maximal cliques of a graph using the Bron-Kerbosch clique detection algorithm: A clique is maximal if it is impossible to enlarge it by adding another vertex from the graph. Note that a maximal clique is not necessarily the biggest clique in the graph.

Examples:

```
graph my_graph <- graph([]);
list<list> var1 <- maximal_cliques_of (my_graph);   // var1
   equals the list of all the maximal cliques as list
```

See also:

biggest_cliques_of,

---

mean

Possible use:

- `mean` (`container`) —> `unknown`

Result:

the mean of all the elements of the operand

Comment:

the elements of the operand are summed (see sum for more details about the sum of container elements) and then the sum value is divided by the number of elements.

Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

Examples:

```
unknown var0 <- mean ([4.5, 3.5, 5.5, 7.0]);    // var0 equals
   5.125
```

See also:

sum,

---

mean_deviation

Possible use:

- `mean_deviation` (`container`) —> `float`

Result:

the deviation from the mean of all the elements of the operand. See Mean_deviation for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- mean_deviation ([4.5, 3.5, 5.5, 7.0]);    // var0
   equals 1.125
```

See also:

mean, standard_deviation,

---

mean_of

Possible use:

- `container` `mean_of` `any expression` —> `unknown`
- `mean_of` (`container` , `any expression`) —> `unknown`

Result:

the mean of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var2 <- [1::2, 3::4, 5::6] mean_of (each);  // var2
    equals 4
```

Examples:

```
unknown var1 <- [1,2] mean_of (each * 10 );     // var1 equals 15
```

See also:

min_of, max_of, sum_of, product_of,

`meanR`

Possible use:

- `meanR` (`container`) —> `unknown`

Result:

returns the mean value of given vector (right-hand operand) in given variable (left-hand operand).

Examples:

```
list<int> X <- [2, 3, 1];
int var1 <- meanR(X);   // var1 equals 2
```

---

median

Possible use:

- median (container) —> unknown

Result:

the median of all the elements of the operand.

Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

Examples:

```
unknown var0 <- median ([4.5, 3.5, 5.5, 3.4, 7.0]);     // var0
   equals 5.0
```

See also:

mean,

---

message

Possible use:

- message (unknown) —> msi.gama.extensions.messaging.GamaMessage

Result:

to be added

---

min

Possible use:

- `min` (`container`) —> `unknown`

Result:

the minimum element found in the operand.

Comment:

the min operator behavior depends on the nature of the operand

Special cases:

- if it is a list of points: min returns the minimum of all points as a point (i.e. the point with the smallest coordinate on the x-axis, in case of equality the point with the smallest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)
- if it is a population of a list of other types: min transforms all elements into integer and returns the minimum of them
- if it is a map, min returns the minimum among the list of all elements value
- if it is a file, min returns the minimum of the content of the file (that is also a container)

- if it is a graph, min returns the minimum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, min returns the minimum of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, min returns the minimum of the list of the geometries
- if it is a matrix of another type, min returns the minimum of the elements transformed into float
- if it is a list of int or float: min returns the minimum of all the elements

```
unknown var0 <- min ([100, 23.2, 34.5]);    // var0 equals 23.2
```

See also:

max,

---

min_of

Possible use:

- `container` `min_of` `any expression` —> `unknown`
- `min_of` (`container` , `any expression`) —> `unknown`

Result:

the minimum value of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil or empty, min_of throws an error
- if the left-operand is a map, the keyword each will contain each value

```
unknown var5 <- [1::2, 3::4, 5::6] min_of (each + 3);   // var5
   equals 5
```

Examples:

```
unknown var1 <- [1,2,4,3,5,7,6,8] min_of (each * 100 );     //
   var1 equals 100
graph g2 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
unknown var3 <- g2 min_of (length(g2 out_edges_of each) );  //
   var3 equals 0
unknown var4 <- (list(node) min_of (round(node(each).location.x))
   ;  // var4 equals 4
```

See also:

max_of,

---

mod

Possible use:

- `int` mod `int` —> `int`
- `mod` (`int` , `int`) —> `int`

Result:

Returns the remainder of the integer division of the left-hand operand by the right-hand operand.

Special cases:

- if operands are float, they are truncated
- if the right-hand operand is equal to zero, raises an exception.

Examples:

```
int var0 <- 40 mod 3;    // var0 equals 1
```

See also:

div,

---

mul

Possible use:

- `mul` (`container`) —> `unknown`

Result:

the product of all the elements of the operand

Comment:

the mul operator behavior depends on the nature of the operand

Special cases:

- if it is a list of points: mul returns the product of all points as a point (each coordinate is the product of the corresponding coordinate of each element)
- if it is a list of other types: mul transforms all elements into integer and multiplies them
- if it is a map, mul returns the product of the value of all elements
- if it is a file, mul returns the product of the content of the file (that is also a container)
- if it is a graph, mul returns the product of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, mul returns the product of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, mul returns the product of the list of the geometries
- if it is a matrix of other types: mul transforms all elements into float and multiplies them
- if it is a list of int or float: mul returns the product of all the elements

```
unknown var0 <- mul ([100, 23.2, 34.5]);     // var0 equals
   80040.0
```

See also:

sum,

nb_cycles

Possible use:

- nb_cycles (graph) —> int

Result:

returns the maximum number of independent cycles in a graph. This number (u) is estimated through the number of nodes (v), links (e) and of sub-graphs (p): u = e - v + p.

Examples:

```
graph graphEpidemio <- graph([]);
int var1 <- nb_cycles(graphEpidemio);   // var1 equals the number
    of cycles in the graph
```

See also:

alpha_index, beta_index, gamma_index, connectivity_index,

---

neighbors_at

Possible use:

- geometry neighbors_at float —> container
- neighbors_at (geometry , float) —> container

Result:

a list, containing all the agents of the same species than the left argument (if it is an agent) located at a distance inferior or equal to the right-hand operand to the left-hand operand (geometry, agent, point).

Comment:

The topology used to compute the neighborhood is the one of the left-operand if this one is an agent; otherwise the one of the agent applying the operator.

Examples:

```
container var0 <- (self neighbors_at (10));      // var0 equals
   all the agents located at a distance lower or equal to 10 to
   the agent applying the operator.
```

See also:

neighbors_of, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_-to, at_distance,

---

neighbors_of

Possible use:

- `graph` `neighbors_of` `unknown` —> `container`
- `neighbors_of` (`graph` , `unknown`) —> `container`
- `topology` `neighbors_of` `agent` —> `container`
- `neighbors_of` (`topology` , `agent`) —> `container`
- `neighbors_of` (`topology`, `geometry`, `float`) —> `container`

Result:

a list, containing all the agents of the same species than the argument (if it is an agent) located at a distance inferior or equal to 1 to the right-hand operand agent considering the left-hand operand topology.

Special cases:

- a list, containing all the agents of the same species than the left argument (if it is an agent) located at a distance inferior or equal to the third argument to the second argument (agent, geometry or point) considering the first operand topology.

```
container var0 <- neighbors_of (topology(self), self,10);   //
   var0 equals all the agents located at a distance lower or
   equal to 10 to the agent applying the operator considering its
    topology.
```

Examples:

```
container var1 <- graphEpidemio neighbors_of (node(3));     //
   var1 equals [node0,node2]
container var2 <- graphFromMap neighbors_of node({12,45});  //
   var2 equals [{1.0,5.0},{34.0,56.0}]
container var3 <- topology(self) neighbors_of self;     // var3
   equals returns all the agents located at a distance lower or
   equal to 1 to the agent applying the operator considering its
   topology.
```

See also:

predecessors_of, successors_of, neighbors_at, closest_to, overlapping, agents_overlapping, agents_inside, agent_closest_to,

---

new_emotion

Possible use:

- `new_emotion` (`string`) —> `emotion`
- `string` `new_emotion` `float` —> `emotion`

- `new_emotion` (`string` , `float`) —> `emotion`
- `string` `new_emotion` `predicate` —> `emotion`
- `new_emotion` (`string` , `predicate`) —> `emotion`
- `new_emotion` (`string`, `float`, `float`) —> `emotion`
- `new_emotion` (`string`, `float`, `predicate`) —> `emotion`
- `new_emotion` (`string`, `float`, `predicate`, `float`) —> `emotion`

Result:

a new emotion with the given properties (name, intensity) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name,about) a new emotion with the given properties (name,intensity,decay) a new emotion with the given properties (name,intensity,about)

Examples:

```
emotion("joy",12.3)
emotion("joy",12.3,eatFood,4)
emotion("joy")
emotion("joy",eatFood)
emotion("joy",12.3,4)
emotion("joy",12.3,eatFood)
```

---

new_folder

Possible use:

- `new_folder` (`string`) —> `file`

Result:

opens an existing repository or create a new folder if it does not exist.

Special cases:

- If the specified string does not refer to an existing repository, the repository is created.
- If the string refers to an existing file, an exception is risen.

Examples:

```
file dirNewT <- new_folder("incl/");     // dirNewT represents the
    repository "../incl/"
                                                             //
   eventually creates the directory ../incl
```

See also:

folder, file,

---

new_predicate

Possible use:

- `new_predicate` (`string`) —> `predicate`
- `string` `new_predicate` `map` —> `predicate`
- `new_predicate` (`string` , `map`) —> `predicate`
- `string` `new_predicate` `bool` —> `predicate`
- `new_predicate` (`string` , `bool`) —> `predicate`
- `string` `new_predicate` `int` —> `predicate`
- `new_predicate` (`string` , `int`) —> `predicate`
- `string` `new_predicate` `float` —> `predicate`
- `new_predicate` (`string` , `float`) —> `predicate`
- `new_predicate` (`string`, `map`, `int`) —> `predicate`
- `new_predicate` (`string`, `map`, `bool`) —> `predicate`
- `new_predicate` (`string`, `map`, `float`) —> `predicate`

Result:

a new predicate with the given properties (name, values, lifetime) a new predicate with the given properties (name) a new predicate with the given properties (name, values) a new predicate with the given is_true (name, is_true) a new predicate with the given is_true (name, lifetime) a new predicate with the given properties (name, values, is_true) a new predicate with the given is_true (name, priority) a new predicate with the given properties (name, values, priority)

Examples:

```
predicate("people to meet", ["time"::10], true)
predicate("people to meet")
predicate("people to meet", people1 )
predicate("hasWater", true)
predicate("hasWater", 10
predicate("people to meet", ["time"::10], true)
predicate("hasWater", 2.0 )
predicate("people to meet", people1, ["time"::10])
```

---

node

Possible use:

- node (unknown) —> unknown
- unknown node float —> unknown
- node (unknown , float) —> unknown

---

nodes

Possible use:

- nodes (container) —> container

---

norm

Possible use:

- norm (point) —> float

Result:

the norm of the vector with the coordinates of the point operand.

Examples:

```
float var0 <- norm({3,4});  // var0 equals 5.0
```

---

not

Same signification as !

---

obj_file

Possible use:

- obj_file (string) —> file

Result:

Constructs a file of type obj. Allowed extensions are limited to obj

---

of

Same signification as .

---

of_generic_species

Possible use:

- container of_generic_species species —> container
- of_generic_species (container , species) —> container

Result:

a list, containing the agents of the left-hand operand whose species is that denoted by the right-hand operand and whose species extends the right-hand operand species

Examples:

```
// species test {}
// species sous_test parent: test {}
container var2 <- [sous_test(0),sous_test(1),test(2),test(3)]
   of_generic_species test;  // var2 equals [sous_test0,
   sous_test1,test2,test3]
container var3 <- [sous_test(0),sous_test(1),test(2),test(3)]
   of_generic_species sous_test;     // var3 equals [sous_test0,
   sous_test1]
container var4 <- [sous_test(0),sous_test(1),test(2),test(3)]
   of_species test;  // var4 equals [test2,test3]
container var5 <- [sous_test(0),sous_test(1),test(2),test(3)]
   of_species sous_test;     // var5 equals [sous_test0,
   sous_test1]
```

See also:

of_species,

`of_species`

Possible use:

- `container of_species species —> container`
- `of_species (container , species) —> container`

Result:

a list, containing the agents of the left-hand operand whose species is the one denoted by the right-hand operand.The expression agents of_species (species self) is equivalent to agents where (species each = species self); however, the advantage of using the first syntax is that the resulting list is correctly typed with the right species, whereas, in the second syntax, the parser cannot determine the species of the agents within the list (resulting in the need to cast it explicitly if it is to be used in an ask statement, for instance).

Special cases:

- if the right operand is nil, of_species returns the right operand

Examples:

```
container var0 <- (self neighbors_at 10) of_species (species (
   self));   // var0 equals all the neighboring agents of the
   same species.
container var1 <- [test(0),test(1),node(1),node(2)] of_species
   test;    // var1 equals [test0,test1]
```

See also:

of_generic_species,

---

one_of

Possible use:

- one_of (container<KeyType,ValueType>) —> ValueType

Result:

one of the values stored in this container at a random key

Comment:

the one_of operator behavior depends on the nature of the operand

Special cases:

- if it is a graph, one_of returns one of the lists of edges
- if it is a file, one_of returns one of the elements of the content of the file (that is also a container)
- if the operand is empty, one_of returns nil

```
```

- if it is a list or a matrix, one_of returns one of the values of the list or of the matrix

```
int i <- any ([1,2,3]);      // i equals 1, 2 or 3
string sMat <- one_of(matrix([["c11","c12","c13"],["c21","c22","
    c23"]]));    // sMat equals "c11","c12","c13", "c21","c22" or "
    c23"
```

- if it is a map, one_of returns one the value of a random pair of the map

```
int im <- one_of ([2::3, 4::5, 6::7]);  // im equals 3, 5 or 7
bool var6 <- [2::3, 4::5, 6::7].values contains im;     // var6
   equals true
```

- if it is a population, one_of returns one of the agents of the population

```
bug b <- one_of(bug);   // Given a previously defined species bug
   , b is one of the created bugs, e.g. bug3
```

See also:

contains,

`or`

Possible use:

- `bool` `or` `any expression` —> `bool`
- `or` (`bool` , `any expression`) —> `bool`

Result:

a bool value, equal to the logical or between the left-hand operand and the right-hand operand.

Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like 1 or 0 is accepted and returns true.

See also:

bool, and, !,

---

or

Possible use:

- predicate or predicate —> predicate
- or (predicate , predicate) —> predicate

Result:

create a new predicate from two others by including them as subintentions. It's an exclusive "or"

Examples:

```
predicate1 or predicate2
```

---

osm_file

Possible use:

- string osm_file map<string,list> —> file
- osm_file (string , map<string,list>) —> file
- osm_file (string, map<string,list>, int) —> file

Result:

opens a file that a is a kind of OSM file with some filtering. opens a file that a is a kind of OSM file with some filtering, forcing the initial CRS to be the one indicated by the second int parameter (see http://spatialreference.org/ref/epsg/). If this int parameter is equal to 0, the data is considered as already projected.

Comment:

The file should have a OSM file extension, cf. file type definition for supported file extensions.The file should have a OSM file extension, cf. file type definition for supported file extensions.

Special cases:

- If the specified string does not refer to an existing OSM file, an exception is risen.
- If the specified string does not refer to an existing OSM file, an exception is risen.

Examples:

```
file myOSMfile <- osm_file("../includes/rouen.osm", ["highway"::[
   "primary","motorway"]]);
file myOSMfile2 <- osm_file("../includes/rouen.osm",["highway"::[
   "primary","motorway"]], 0);
```

See also:

file,

out_degree_of

Possible use:

- graph out_degree_of unknown —> int
- out_degree_of (graph , unknown) —> int

Result:

returns the out degree of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap out_degree_of (node(3));    // var1
   equals 4
```

See also:

in_degree_of, degree_of,

---

out_edges_of

Possible use:

- graph out_edges_of unknown —> container
- out_edges_of (graph , unknown) —> container

Result:

returns the list of the out-edges of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
container var1 <- graphFromMap out_edges_of (node(3));  // var1
   equals 3
```

See also:

in_edges_of,

---

overlapping

Possible use:

- `container<agent>` `overlapping` `geometry` —> `list<geometry>`
- `overlapping` (`container<agent>` , `geometry`) —> `list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), overlapping the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] overlapping(self);   //
   var0 equals return the agents among ag1, ag2 and ag3 that
   overlap the shape of the agent applying the operator.
(species1 + species2) overlapping self
```

See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, agents_-overlapping,

---

overlaps

Possible use:

- `geometry` `overlaps` `geometry` —> `bool`
- `overlaps` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) overlaps the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.
- if one operand is a point, returns true if the point is included in the geometry

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) overlaps polyline
   ([{15,15},{25,25}]);  // var0 equals true
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polygon([{15,15},{15,25},{25,25},{25,15}]);    // var1 equals
   true
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   {25,25};   // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polygon([{35,35},{35,45},{45,45},{45,35}]);    // var3 equals
   false
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polyline([{10,10},{20,20}]);   // var4 equals true
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   {15,15};   // var5 equals true
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polygon([{0,0},{0,30},{30,30}, {30,0}]);   // var6 equals true
```

```
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polygon([{15,15},{15,25},{25,25},{25,15}]);     // var7 equals
   true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
   polygon([{10,20},{20,20},{20,30},{10,30}]);     // var8 equals
   true
```

See also:

disjoint_from, crosses, intersects, partially_overlaps, touches,

---

pair

Possible use:

- `pair` (`any`) —> `pair`

Result:

Casts the operand into the type pair

---

partially_overlaps

Possible use:

- `geometry` `partially_overlaps` `geometry` —> `bool`
- `partially_overlaps` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) partially overlaps the right-geometry (or agent/point).

Comment:

if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) partially_overlaps
   polyline([{15,15},{25,25}]);    // var0 equals true
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polygon([{15,15},{15,25},{25,25},{25,15}]);
     // var1 equals true
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps {25,25};     // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polygon([{35,35},{35,45},{45,45},{45,35}]);
     // var3 equals false
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polyline([{10,10},{20,20}]);     // var4
   equals false
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps {15,15};     // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polygon([{0,0},{0,30},{30,30}, {30,0}]);
     // var6 equals false
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polygon([{15,15},{15,25},{25,25},{25,15}]);
     // var7 equals true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}])
   partially_overlaps polygon([{10,20},{20,20},{20,30},{10,30}]);
     // var8 equals false
```

See also:

disjoint_from, crosses, overlaps, intersects, touches,

`path`

Possible use:

- `path` (`any`) —> `path`

Result:

Casts the operand into the type path

`path_between`

Possible use:

- `topology` `path_between` `container<geometry>` —> `path`
- `path_between` (`topology` , `container<geometry>`) —> `path`
- `path_between` (`list<agent>`, `geometry`, `geometry`) —> `path`
- `path_between` (`graph`, `geometry`, `geometry`) —> `path`

Result:

The shortest path between two objects according to set of cells The shortest path between a list of two objects in a graph

Examples:

```
path var0 <- my_topology path_between [ag1, ag2];   // var0
   equals A path between ag1 and ag2
path var1 <- path_between (cell_grid where each.is_free, ag1, ag2
   );     // var1 equals A path between ag1 and ag2 passing
   through the given cell_grid agents
path var2 <- path_between (my_graph, ag1, ag2);     // var2
   equals A path between ag1 and ag2
```

See also:

towards, direction_to, distance_between, direction_between, path_to, distance_to,

---

path_to

Possible use:

- geometry **path_to** geometry —> path
- **path_to** (geometry , geometry) —> path
- point **path_to** point —> path
- **path_to** (point , point) —> path

Result:

A path between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

Examples:

```
path var0 <- ag1 path_to ag2;   // var0 equals the path between
   ag1 and ag2 considering the topology of the agent applying the
    operator
```

See also:

towards, direction_to, distance_between, direction_between, path_between, distance_to,

---

`paths_between`

Possible use:

- `paths_between (graph, pair, int) —> list<path>`

Result:

The K shortest paths between a list of two objects in a graph

Examples:

```
list<path> var0 <- paths_between(my_graph, ag1:: ag2, 2);   //
   var0 equals the 2 shortest paths (ordered by length) between
   ag1 and ag2
```

`percent_absolute_deviation`

Possible use:

- `list<float> percent_absolute_deviation list<float> —> float`
- `percent_absolute_deviation (list<float> , list<float>) —> float`

Result:

percent absolute deviation indicator for 2 series of values: percent_absolute_deviation(list_vals_observe,list_vals_sim)

Examples:

```
percent_absolute_deviation
   ([200,300,150,150,200],[250,250,100,200,200])
```

---

pgm_file

Possible use:

- pgm_file (string) —> file

Result:

Constructs a file of type pgm. Allowed extensions are limited to pgm

---

plan

Possible use:

- container<geometry> plan float —> geometry
- plan (container<geometry> , float) —> geometry

Result:

A polyline geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}

- if the operand is composed of a single point, returns a point geometry.

Examples:

```
geometry var0 <- polyplan([{0,0}, {0,10}, {10,10}, {10,0}],10);
      // var0 equals a polyline geometry composed of the 4 points
    with a depth of 10.
```

See also:

around, circle, cone, link, norm, point, polygone, rectangle, square, triangle,

---

point

Possible use:

- `int` **`point`** `float` —> `point`
- **`point`** (`int` , `float`) —> `point`
- `float` **`point`** `int` —> `point`
- **`point`** (`float` , `int`) —> `point`
- `int` **`point`** `int` —> `point`
- **`point`** (`int` , `int`) —> `point`
- `float` **`point`** `float` —> `point`
- **`point`** (`float` , `float`) —> `point`
- **`point`** (`float`, `int`, `int`) —> `point`
- **`point`** (`int`, `int`, `float`) —> `point`
- **`point`** (`int`, `int`, `int`) —> `point`
- **`point`** (`float`, `float`, `float`) —> `point`
- **`point`** (`int`, `float`, `float`) —> `point`
- **`point`** (`float`, `float`, `int`) —> `point`
- **`point`** (`float`, `int`, `float`) —> `point`

Result:

internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y, z} instead. internal use only. Use the standard construction {x,y} instead. internal use only. Use the standard construction {x,y} instead.

---

points_at

Possible use:

- `int` `points_at` `float` —> `list<point>`
- `points_at` (`int` , `float`) —> `list<point>`

Result:

A list of left-operand number of points located at a the right-operand distance to the agent location.

Examples:

```
list<point> var0 <- 3 points_at(20.0);  // var0 equals returns [
    pt1, pt2, pt3] with pt1, pt2 and pt3 located at a distance of
    20.0 to the agent location
```

See also:

any_location_in, any_point_in, closest_points_with, farthest_point_to,

---

points_on

Possible use:

- geometry points_on float —> container
- points_on (geometry , float) —> container

Result:

A list of points of the operand-geometry distant from each other to the float right-operand .

Examples:

```
container var0 <-  square(5) points_on(2);  // var0 equals a list
    of points belonging to the exterior ring of the square
   distant from each other of 2.
```

See also:

closest_points_with, farthest_point_to, points_at,

---

poisson

Possible use:

- poisson (float) —> int

Result:

A value from a random variable following a Poisson distribution (with the positive expected number of occurence lambda as operand).

Comment:

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event, cf. Poisson distribution on Wikipedia.

Examples:

```
int var0 <- poisson(3.5);   // var0 equals a random positive
   integer
```

See also:

binomial, gauss,

---

polygon

Possible use:

- polygon (container<agent>) —> geometry

Result:

A polygon geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry
- if the operand is composed of 2 points, returns a polyline geometry.

Examples:

```
geometry var0 <- polygon([{0,0}, {0,10}, {10,10}, {10,0}]);
   // var0 equals a polygon geometry composed of the 4 points.
```

See also:

around, circle, cone, line, link, norm, point, polyline, rectangle, square, triangle,

---

polyhedron

Possible use:

- `container<geometry> polyhedron float —> geometry`
- `polyhedron (container<geometry> , float) —> geometry`

Result:

A polyhedron geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry
- if the operand is composed of 2 points, returns a polyline geometry.

Examples:

```
geometry var0 <- polyhedron([{0,0}, {0,10}, {10,10}, {10,0}],10);
      // var0 equals a polygon geometry composed of the 4 points
   and of depth 10.
```

See also:

around, circle, cone, line, link, norm, point, polyline, rectangle, square, triangle,

---

`polyline`

Same signification as line

`polyplan`

Same signification as plan

`predecessors_of`

Possible use:

- `graph` `predecessors_of` `unknown` —> `container`
- `predecessors_of` (`graph` , `unknown`) —> `container`

Result:

returns the list of predecessors (i.e. sources of in edges) of the given vertex (right-hand operand) in the given graph (left-hand operand)

Examples:

```
container var1 <- graphEpidemio predecessors_of ({1,5});    //
   var1 equals []
container var2 <- graphEpidemio predecessors_of node({34,56});
   // var2 equals [{12;45}]
```

See also:

neighbors_of, successors_of,

`predicate`

Possible use:

- `predicate` (`any`) —> `predicate`

Result:

Casts the operand into the type predicate

`predict`

Possible use:

- `regression` `predict` `list<float>` —> `float`
- `predict` (`regression` , `list<float>`) —> `float`

Result:

returns the value predict by the regression parameters for a given instance. Usage: predict(regression, instance)

Examples:

```
predict(my_regression, [1,2,3]
```

---

product

Same signification as mul

---

product_of

Possible use:

- container product_of any expression —> unknown
- product_of (container , any expression) —> unknown

Result:

the product of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var2 <- [1::2, 3::4, 5::6] product_of (each);    // var2
   equals 48
```

Examples:

```
unknown var1 <- [1,2] product_of (each * 10 );  // var1 equals
   200
```

See also:

min_of, max_of, sum_of, mean_of,

---

promethee_DM

Possible use:

- `list<list>` `promethee_DM` `list<map<string,object>>` —> `int`
- `promethee_DM` (`list<list>` , `list<map<string,object>>`) —> `int`

Result:

The index of the best candidate according to the Promethee II method. This method is based on a comparison per pair of possible candidates along each criterion: all candidates are compared to each other by pair and ranked. More information about this method can be found in [http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6VCT-4VF56TV-1&_user=10&_coverDate=01%2F01%2F2010&_rdoc=1&_fmt=high&_orig=search&_sort=d&_docanchor=&view=c&_searchStrId=1389284642&_rerunOrigin=google&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=d334de2a4e0d6281199a39857648cd36 Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research(2009)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fours elements: a name, a weight, a preference value (p) and an indifference value (q). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant.

Special cases:

- returns -1 is the list of candidates is nil or empty

Examples:

```
int var0 <- promethee_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["
   name"::"utility", "weight" :: 2.0,"p"::0.5, "q"::0.0, "s"
   ::1.0, "maximize" :: true],["name"::"price", "weight" :: 1.0,"
   p"::0.5, "q"::0.0, "s"::1.0, "maximize" :: false]]);  // var0
   equals 1
```

See also:

weighted_means_DM, electre_DM, evidence_theory_DM,

---

property_file

Possible use:

- `property_file` (`string`) —> `file`

Result:

Constructs a file of type property. Allowed extensions are limited to properties

---

pyramid

Possible use:

- `pyramid` (`float`) —> `geometry`

Result:

A square geometry which side size is given by the operand.

Comment:

the center of the pyramid is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- pyramid(5);    // var0 equals a geometry as a
   square with side_size = 5.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, square,

---

R_correlation

Same signification as corR

R_file

Possible use:

- `R_file` (`string`) —> `file`

Result:

Constructs a file of type R. Allowed extensions are limited to r

---

`R_mean`

Same signification as meanR

`range`

Possible use:

- `range` (`int`) —> `container`
- `int` `range` `int` —> `container`
- `range` (`int` , `int`) —> `container`
- `range` (`int`, `int`, `int`) —> `container`

Result:

Allows to build a list of int representing all contiguous values from zero to the argument. The range can be increasing or decreasing. Passing 0 will return a singleton list with 0 Allows to build a list of int representing all contiguous values from the first to the second argument, using the step represented by the third argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value. Passing a step of 0 will result in an exception. Attempting to build infinite ranges (e.g. end > start with a negative step) will similarly not be accepted and yield an exception Allows to build a list of int representing all contiguous values from the first to the second argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value

---

read

Possible use:

- read (string) —> unknown

Result:

Reads an attribute of the agent. The attribute's name is specified by the operand.

Examples:

```
unknown agent_name <- read ('name');     // agent_name equals
   reads the 'name' variable of agent then assigns the returned
   value to the 'agent_name' variable.
```

---

rectangle

Possible use:

- rectangle (point) —> geometry
- float rectangle float —> geometry
- rectangle (float , float) —> geometry
- point rectangle point —> geometry
- rectangle (point , point) —> geometry

Result:

A rectangle geometry which side sizes are given by the operands.

Comment:

the center of the rectangle is by default the location of the current agent in which has been called this operator.the center of the rectangle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- rectangle(10, 5);  // var0 equals a geometry as
   a rectangle with width = 10 and height = 5.
geometry var1 <- rectangle({2.0,6.0}, {6.0,20.0});  // var1
   equals a geometry as a rectangle with {2.0,6.0} as the upper-
   left corner, {6.0,20.0} as the lower-right corner.
geometry var2 <- rectangle({10, 5});    // var2 equals a geometry
    as a rectangle with width = 10 and height = 5.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, square, triangle,

---

reduced_by

Same signification as -

---

regression

Possible use:

- `regression` (any) —> `regression`

Result:

Casts the operand into the type regression

`remove_duplicates`

Possible use:

- `remove_duplicates` (`container`) —> `container`

Result:

produces a set from the elements of the operand (i.e. a list without duplicated elements)

Special cases:

- if the operand is nil, remove_duplicates returns nil
- if the operand is a graph, remove_duplicates returns the set of nodes
- if the operand is a matrix, remove_duplicates returns a matrix without duplicated row
- if the operand is a map, remove_duplicates returns the set of values without duplicate

```
container var1 <- remove_duplicates([1::3,2::4,3::3,5::7]);
   // var1 equals [3,4,7]
```

Examples:

```
container var0 <- remove_duplicates([3,2,5,1,2,3,5,5,5]);   //
   var0 equals [3,2,5,1]
```

---

remove_node_from

Possible use:

- geometry remove_node_from graph —> graph
- remove_node_from (geometry , graph) —> graph

Result:

removes a node from a graph.

Comment:

all the edges containing this node are also removed.

Examples:

```
graph var0 <- node(0) remove_node_from graphEpidemio;   // var0
   equals the graph without node(0)
```

---

replace

Possible use:

- replace (string, string, string) —> string

Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding the second operand by the third operand

Examples:

```
string var0 <- replace('to be or not to be,that is the question
   ','to', 'do');   // var0 equals 'do be or not do be,that is
   the question'
```

See also:

replace_regex,

---

replace_regex

Possible use:

- `replace_regex` (string, string, string) —> string

Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding to the regular expression given in the second operand by the third operand

Examples:

```
string var0 <- replace_regex("colour, color", "colou?r", "col");
      // var0 equals 'col, col'
```

See also:

replace,

`reverse`

Possible use:

- `reverse` (`container<KeyType,ValueType>`) —> `container`
- `reverse` (`string`) —> `string`

Result:

the operand elements in the reversed order in a copy of the operand.

Comment:

the reverse operator behavior depends on the nature of the operand

Special cases:

- if it is a file, reverse returns a copy of the file with a reversed content
- if it is a population, reverse returns a copy of the population with elements in the reversed order
- if it is a graph, reverse returns a copy of the graph (with all edges and vertexes), with all of the edges reversed
- if it is a list, reverse returns a copy of the operand list with elements in the reversed order

```
container var0 <- reverse ([10,12,14]);     // var0 equals [14,
   12, 10]
```

- if it is a map, reverse returns a copy of the operand map with each pair in the reversed order (i.e. all keys become values and values become keys)

```
container var1 <- reverse (['k1'::44, 'k2'::32, 'k3'::12]);
   // var1 equals [12::'k3',  32::'k2', 44::'k1']
```

- if it is a matrix, reverse returns a new matrix containing the transpose of the operand.

```
container var2 <- reverse(matrix([["c11","c12","c13"],["c21","c22
   ","c23"]]));   // var2 equals matrix([["c11","c21"],["c12","
   c22"],["c13","c23"]])
```

- if it is a string, reverse returns a new string with characters in the reversed order

```
string var3 <- reverse ('abcd');    // var3 equals 'dcba'
```

---

rewire_n

Possible use:

- graph rewire_n int —> graph
- rewire_n (graph , int) —> graph

Result:

rewires the given count of edges.

Comment:

If there are too many edges, all the edges will be rewired.

Examples:

```
graph var1 <- graphEpidemio rewire_n 10;    // var1 equals the
   graph with 3 edges rewired
```

`rgb`

Possible use:

- `rgb` **`rgb`** `int` —> `rgb`
- **`rgb`** (`rgb` , `int`) —> `rgb`
- `rgb` **`rgb`** `float` —> `rgb`
- **`rgb`** (`rgb` , `float`) —> `rgb`
- `string` **`rgb`** `int` —> `rgb`
- **`rgb`** (`string` , `int`) —> `rgb`
- **`rgb`** (`int`, `int`, `int`) —> `rgb`
- **`rgb`** (`int`, `int`, `int`, `int`) —> `rgb`
- **`rgb`** (`int`, `int`, `int`, `float`) —> `rgb`

Result:

Returns a color defined by red, green, blue components and an alpha blending value.

Special cases:

- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0 and 255)
- It can be used with a color and an alpha between 0 and 255
- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0.0 and 1.0)
- It can be used with a color and an alpha between 0 and 1
- It can be used with r=red, g=green, b=blue, each between 0 and 255
- It can be used with a name of color and alpha (between 0 and 255)

Examples:

```
rgb var0 <- rgb (255,0,0,125);  // var0 equals a light red color
rgb var2 <- rgb(rgb(255,0,0),125);  // var2 equals a light red
   color
rgb var3 <- rgb (255,0,0,0.5);  // var3 equals a light red color
rgb var4 <- rgb(rgb(255,0,0),0.5);  // var4 equals a light red
   color
rgb var5 <- rgb (255,0,0);  // var5 equals #red
rgb var6 <- rgb ("red");    // var6 equals rgb(255,0,0)
```

See also:

hsb,

rgb_to_xyz

Possible use:

- `rgb_to_xyz (file) —> list<point>`

Result:

A list of point corresponding to RGB value of an image (r:x , g:y, b:z)

Examples:

```
list<point> var0 <- rgb_to_xyz(texture);    // var0 equals a list
    of points
```

`rnd`

Possible use:

- `rnd` (`point`) —> `point`
- `rnd` (`int`) —> `int`
- `rnd` (`float`) —> `float`
- `float` `rnd` `float` —> `float`
- `rnd` (`float` , `float`) —> `float`
- `point` `rnd` `point` —> `point`
- `rnd` (`point` , `point`) —> `point`
- `int` `rnd` `int` —> `int`
- `rnd` (`int` , `int`) —> `int`
- `rnd` (`point`, `point`, `float`) —> `point`
- `rnd` (`float`, `float`, `float`) —> `float`
- `rnd` (`int`, `int`, `int`) —> `int`

Result:

a random integer in the interval [0, operand]

Comment:

to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision

Special cases:

- if the operand is a point, returns a point with three random float ordinates, each in the interval [0, ordinate of argument]

- if the operand is a float, returns an uniformly distributed float random number in [0.0, to]

Examples:

```
point var0 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0}, 1);    // var0
   equals a point with x = 2.0, y equal to 2.0, 3.0 or 4.0 and z
   between 0.0 and 10.0 every 1.0
float var1 <- rnd (2.0, 4.0);   // var1 equals a float number
   between 2.0 and 4.0
point var2 <- rnd ({2.5,3, 0.0});   // var2 equals {x,y} with x
   in [0.0,2.0], y in [0.0,3.0], z = 0.0
float var3 <- rnd (2.0, 4.0, 0.5);  // var3 equals a float number
    between 2.0 and 4.0 every 0.5
int var4 <- rnd (2);    // var4 equals 0, 1 or 2
float var5 <- rnd (1000) / 1000;    // var5 equals a float
   between 0 and 1 with a precision of 0.001
float var6 <- rnd(3.4);     // var6 equals a random float between
    0.0 and 3.4
int var7 <- rnd (2, 12, 4);     // var7 equals 2, 6 or 10
point var8 <- rnd ({2.0, 4.0}, {2.0, 5.0, 10.0});   // var8
   equals a point with x = 2.0, y between 2.0 and 4.0 and z
   between 0.0 and 10.0
int var9 <- rnd (2, 4);     // var9 equals 2, 3 or 4
```

See also:

flip,

---

rnd_choice

Possible use:

- `rnd_choice` (`container`) —> `int`

Result:

returns an index of the given list with a probability following the (normalized) distribution described in the list (a form of lottery)

Examples:

```
int var0 <- rnd_choice([0.2,0.5,0.3]);  // var0 equals 2/10
   chances to return 0, 5/10 chances to return 1, 3/10 chances to
    return 2
```

See also:

rnd,

---

rnd_color

Possible use:

- rnd_color (int) —> rgb

Result:

rgb color

Comment:

Return a random color equivalent to rgb(rnd(operand),rnd(operand),rnd(operand))

Examples:

```
rgb var0 <- rnd_color(255);     // var0 equals a random color,
   equivalent to rgb(rnd(255),rnd(255),rnd(255))
```

See also:

rgb, hsb,

---

rotated_by

Possible use:

- `geometry rotated_by int —> geometry`
- `rotated_by (geometry , int) —> geometry`
- `geometry rotated_by float —> geometry`
- `rotated_by (geometry , float) —> geometry`
- `rotated_by (geometry, float, point) —> geometry`

Result:

A geometry resulting from the application of a rotation by the right-hand operand angles (degree) along the three axis (x,y,z) to the left-hand operand (geometry, agent, point) A geometry resulting from the application of a rotation by the right-hand operand angle (degree) to the left-hand operand (geometry, agent, point)

Comment:

the right-hand operand can be a float or a int

Examples:

```
geometry var0 <- rotated_by(pyramid(10),45, {1,0,0});   // var0
   equals the geometry resulting from a 45 degrees rotation along
    the {1,0,0} vector to the geometry of the agent applying the
   operator.
geometry var1 <- self rotated_by 45;    // var1 equals the
   geometry resulting from a 45 degrees rotation to the geometry
   of the agent applying the operator.
```

See also:

transformed_by, translated_by,

---

round

Possible use:

- `round` (`int`) —> `int`
- `round` (`float`) —> `int`
- `round` (`point`) —> `point`

Result:

Returns the rounded value of the operand.

Special cases:

- if the operand is an int, round returns it

Examples:

```
int var0 <- round (0.51);   // var0 equals 1
int var1 <- round (100.2);  // var1 equals 100
int var2 <- round(-0.51);   // var2 equals -1
point var3 <- {12345.78943,  12345.78943, 12345.78943}
   with_precision 2;    // var3 equals
   {12345.79,12345.79,12345.79}
```

See also:

int, with_precision, round,

---

row_at

Possible use:

- matrix row_at int —> list
- row_at (matrix , int) —> list

Result:

returns the row at a num_line (right-hand operand)

Examples:

```
list var0 <- matrix([["el11","el12","el13"],["el21","el22","el23"
   ],["el31","el32","el33"]]) row_at 2;   // var0 equals ["el13
   ","el23","el33"]
```

See also:

column_at, columns_list,

---

rows_list

Possible use:

- rows_list (matrix) —> list<list>

Result:

returns a list of the rows of the matrix, with each row as a list of elements

Examples:

```
list<list> var0 <- rows_list(matrix([["el11","el12","el13"],["
   el21","el22","el23"],["el31","el32","el33"]])); // var0
   equals [["el11","el21","el31"],["el12","el22","el32"],["el13
   ","el23","el33"]]
```

See also:

columns_list,

---

sample

Possible use:

- `sample` (any expression) —> `string`
- `string` `sample` any expression —> `string`
- `sample` (`string` , any expression) —> `string`
- `sample` (`container`, `int`, `bool`) —> `container`
- `sample` (`container`, `int`, `bool`, `container`) —> `container`

Result:

takes a sample of the specified size from the elements of x using either with or without replacement takes a sample of the specified size from the elements of x using either with or without replacement with given weights

Examples:

```
container var0 <- sample([2,10,1],2,false);      // var0 equals
   [1,2]
container var1 <- sample([2,10,1],2,false,[0.1,0.7,0.2]);   //
   var1 equals [10,2]
```

---

scaled_by

Same signification as *

---

scaled_to

Possible use:

- geometry scaled_to point —> geometry
- scaled_to (geometry , point) —> geometry

Result:

allows to restrict the size of a geometry so that it fits in the envelope {width, height, depth} defined by the second operand

Examples:

```
geometry var0 <- shape scaled_to {10,10};   // var0 equals a
   geometry corresponding to the geometry of the agent applying
   the operator scaled so that it fits a square of 10x10
```

---

select

Same signification as where

---

set_about

Possible use:

- emotion set_about predicate —> emotion
- set_about (emotion , predicate) —> emotion

Result:

change the about value of the given emotion

Examples:

```
emotion about predicate1
```

---

set_decay

Possible use:

- emotion set_decay float —> emotion
- set_decay (emotion , float) —> emotion

Result:

change the decay value of the given emotion

Examples:

```
emotion set_decay 12
```

`set_intensity`

Possible use:

- `emotion` `set_intensity` `float` —> `emotion`
- `set_intensity` (`emotion` , `float`) —> `emotion`

Result:

change the intensity value of the given emotion

Examples:

```
emotion set_intensity 12
```

`set_truth`

Possible use:

- `predicate` `set_truth` `bool` —> `predicate`
- `set_truth` (`predicate` , `bool`) —> `predicate`

Result:

change the is_true value of the given predicate

Examples:

```
predicate set_truth false
```

---

set_z

Possible use:

- geometry set_z container<float> —> geometry
- set_z (geometry , container<float>) —> geometry
- set_z (geometry, int, float) —> geometry

Result:

Sets the z ordinate of the n-th point of a geometry to the value provided by the third argument

Examples:

```
loop i from: 0 to: length(shape.points) - 1{set shape <-  set_z (
   shape, i, 3.0);}
shape <- triangle(3) set_z [5,10,14];
```

---

shape_file

Possible use:

- shape_file (string) —> file

Result:

Constructs a file of type shape. Allowed extensions are limited to shp

---

shuffle

Possible use:

- shuffle (string) —> string
- shuffle (matrix) —> matrix
- shuffle (container) —> container

Result:

The elements of the operand in random order.

Special cases:

- if the operand is empty, returns an empty list (or string, matrix)

Examples:

```
string var0 <- shuffle ('abc');     // var0 equals 'bac' (for
   example)
matrix var1 <- shuffle (matrix([["c11","c12","c13"],["c21","c22",
   "c23"]]));     // var1 equals matrix([["c12","c21","c11"],["
   c13","c22","c23"]]) (for example)
container var2 <- shuffle ([12, 13, 14]);   // var2 equals
   [14,12,13] (for example)
```

See also:

reverse,

---

signum

Possible use:

- signum (float) —> int

Result:

Returns -1 if the argument is negative, +1 if it is positive, 0 if it is equal to zero or not a number

Examples:

```
int var0 <- signum(-12);    // var0 equals -1
int var1 <- signum(14);     // var1 equals 1
int var2 <- signum(0);  // var2 equals 0
```

---

simple_clustering_by_distance

Possible use:

- container<agent> simple_clustering_by_distance float —> list<list<agent>>
- simple_clustering_by_distance (container<agent> , float) —> list<list<agent>>

Result:

A list of agent groups clustered by distance considering a distance min between two groups.

Examples:

```
list<list<agent>> var0 <- [ag1, ag2, ag3, ag4, ag5]
   simpleClusteringByDistance 20.0;    // var0 equals for example
   , can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

See also:

hierarchical_clustering,

---

simple_clustering_by_envelope_distance

Same signification as simple_clustering_by_distance

---

simplification

Possible use:

- `geometry` `simplification` `float` —> `geometry`
- `simplification` (`geometry` , `float`) —> `geometry`

Result:

A geometry corresponding to the simplification of the operand (geometry, agent, point) considering a tolerance distance.

Comment:

The algorithm used for the simplification is Douglas-Peucker

Examples:

```
geometry var0 <- self simplification 0.1;   // var0 equals the
   geometry resulting from the application of the Douglas-Peuker
   algorithm on the geometry of the agent applying the operator
   with a tolerance distance of 0.1.
```

---

sin

Possible use:

- `sin (int) —> float`
- `sin (float) —> float`

Result:

Returns the value (in [-1,1]) of the sinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- sin (0);  // var0 equals 0.0
float var1 <- sin(360);     // var1 equals 0.0
```

See also:

cos, tan,

---

sin_rad

Possible use:

- `sin_rad (float) —> float`

Result:

Returns the value (in [-1,1]) of the sinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- sin(360);     // var0 equals 0.0
```

See also:

cos, tan,

`skeletonize`

Possible use:

- `skeletonize` (`geometry`) —> `list<geometry>`

Result:

A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent)

Examples:

```
list<geometry> var0 <- skeletonize(self);    // var0 equals the
    list of geometries corresponding to the skeleton of the
    geometry of the agent applying the operator.
```

---

skewness

Possible use:

- skewness (list) —> float

Result:

returns skewness value computed from the operand list of values

Special cases:

- if the length of the list is lower than 3, returns NaN

Examples:

```
skewness ([1,2,3,4,5])
```

---

skill

Possible use:

- skill (any) —> skill

Result:

Casts the operand into the type skill

---

smooth

Possible use:

- geometry smooth float —> geometry
- smooth (geometry , float) —> geometry

Result:

Returns a 'smoothed' geometry, where straight lines are replaces by polynomial (bicubic) curves. The first parameter is the original geometry, the second is the 'fit' parameter which can be in the range 0 (loose fit) to 1 (tightest fit).

Examples:

```
geometry var0 <- smooth(square(10), 0.0);   // var0 equals a '
   rounded' square
```

---

solid

Same signification as without_holes

---

sort

Same signification as sort_by

---

sort_by

Possible use:

- `container sort_by any expression —> container`
- `sort_by (container , any expression) —> container`

Result:

Returns a list, containing the elements of the left-hand operand sorted in ascending order by the value of the right-hand operand when it is evaluated on them.

Comment:

the left-hand operand is casted to a list before applying the operator. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

Special cases:

- if the left-hand operand is nil, sort_by throws an error

Examples:

```
container var0 <- [1,2,4,3,5,7,6,8] sort_by (each);     // var0
   equals [1,2,3,4,5,6,7,8]
container var2 <- g2 sort_by (length(g2 out_edges_of each) );
   // var2 equals [node9, node7, node10, node8, node11, node6,
   node5, node4]
container var3 <- (list(node) sort_by (round(node(each).location.
   x));   // var3 equals [node5, node1, node0, node2, node3]
```

```
container var4 <- [1::2, 5::6, 3::4] sort_by (each);     // var4
    equals [2, 4, 6]
```

See also:

group_by,

---

source_of

Possible use:

- `graph` `source_of` `unknown` —> `unknown`
- `source_of` (`graph` , `unknown`) —> `unknown`

Result:

returns the source of the edge (right-hand operand) contained in the graph given in left-hand operand.

Special cases:

- if the lef-hand operand (the graph) is nil, throws an Exception

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"
    ::edge,"vertices_specy"::node,"size"::3,"m"::5] );
unknown var1 <- graphEpidemio source_of(edge(3));    // var1
    equals node1
graph graphFromMap <-  as_edge_graph
    ([{1,5}::{12,45},{12,45}::{34,56}]);
point var3 <- graphFromMap source_of(link({1,5}::{12,45}));
    // var3 equals {1,5}
```

See also:

target_of,

---

spatial_graph

Possible use:

- `spatial_graph` (`container`) —> `graph`

Result:

allows to create a spatial graph from a container of vertices, without trying to wire them. The container can be empty. Emits an error if the contents of the container are not geometries, points or agents

See also:

graph,

---

species

Possible use:

- `species` (`unknown`) —> `species`

Result:

casting of the operand to a species.

Special cases:

- if the operand is nil, returns nil;
- if the operand is an agent, returns its species;
- if the operand is a string, returns the species with this name (nil if not found);
- otherwise, returns nil

Examples:

```
species var0 <- species(self);  // var0 equals the species of the
    current agent
species var1 <- species('node');    // var1 equals node
species var2 <- species([1,5,9,3]);     // var2 equals nil
species var3 <- species(node1);     // var3 equals node
```

---

species_of

Same signification as species

---

sphere

Possible use:

- `sphere` (`float`) —> `geometry`

Result:

A sphere geometry which radius is equal to the operand.

Comment:

the centre of the sphere is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- sphere(10);    // var0 equals a geometry as a
   circle of radius 10 but displays a sphere.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

`split_at`

Possible use:

- `geometry` `split_at` `point` —> `list<geometry>`
- `split_at` (`geometry` , `point`) —> `list<geometry>`

Result:

The two part of the left-operand lines split at the given right-operand point

Special cases:

- if the left-operand is a point or a polygon, returns an empty list

Examples:

```
list<geometry> var0 <- polyline([{1,2},{4,6}]) split_at {7,6};
   // var0 equals [polyline([{1.0,2.0},{7.0,6.0}]), polyline
   ([{7.0,6.0},{4.0,6.0}])]
```

---

split_geometry

Possible use:

- `geometry` **`split_geometry`** `point` —> `list<geometry>`
- **`split_geometry`** (`geometry` , `point`) —> `list<geometry>`
- `geometry` **`split_geometry`** `float` —> `list<geometry>`
- **`split_geometry`** (`geometry` , `float`) —> `list<geometry>`
- **`split_geometry`** (`geometry`, `int`, `int`) —> `list<geometry>`

Result:

A list of geometries that result from the decomposition of the geometry according to a grid with the given number of rows and columns (geometry, nb_cols, nb_rows) A list of geometries that result from the decomposition of the geometry by rectangle cells of the given dimension (geometry, {size_x, size_y}) A list of geometries that result from the decomposition of the geometry by square cells of the given side size (geometry, size)

Examples:

```
list<geometry> var0 <- to_rectangles(self, 10,20);  // var0
   equals the list of the geometries corresponding to the
   decomposition of the geometry of the agent applying the
   operator
list<geometry> var1 <- to_rectangles(self, {10.0, 15.0});   //
   var1 equals the list of the geometries corresponding to the
   decomposition of the geometry by rectangles of size 10.0, 15.0
```

```
list<geometry> var2 <- to_squares(self, 10.0);  // var2 equals
   the list of the geometries corresponding to the decomposition
   of the geometry by squares of side size 10.0
```

---

split_lines

Possible use:

- split_lines (container<geometry>) —> list<geometry>

Result:

A list of geometries resulting after cutting the lines at their intersections.

Examples:

```
list<geometry> var0 <- split_lines([line([{0,10}, {20,10}]), line
   ([{0,10}, {20,10}])]);     // var0 equals a list of four
   polylines: line([{0,10}, {10,10}]), line([{10,10}, {20,10}]),
   line([{10,0}, {10,10}]) and line([{10,10}, {10,20}])
```

---

split_with

Possible use:

- string split_with string —> container
- split_with (string , string) —> container

Result:

Returns a list containing the sub-strings (tokens) of the left-hand operand delimited by each of the characters of the right-hand operand.

Comment:

Delimiters themselves are excluded from the resulting list.

Examples:

```
container var0 <- 'to be or not to be,that is the question'
   split_with ' ,';     // var0 equals ['to','be','or','not','to
   ','be','that','is','the','question']
```

---

sqrt

Possible use:

- sqrt (int) —> float
- sqrt (float) —> float

Result:

Returns the square root of the operand.

Special cases:

- if the operand is negative, an exception is raised

Examples:

```
float var0 <- sqrt(4);  // var0 equals 2.0
float var1 <- sqrt(4);  // var1 equals 2.0
```

---

square

Possible use:

- square (float) —> geometry

Result:

A square geometry which side size is equal to the operand.

Comment:

the centre of the square is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- square(10);    // var0 equals a geometry as a
   square of side size 10.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

`squircle`

Possible use:

- `float` `squircle` `float` —> `geometry`
- `squircle` (`float` , `float`) —> `geometry`

Result:

A mix of square and circle geometry (see : http://en.wikipedia.org/wiki/Squircle), which side size is equal to the first operand and power is equal to the second operand

Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the side operand is lower or equal to 0.

Examples:

```
geometry var0 <- squircle(4,4);      // var0 equals a geometry as
   a squircle of side 4 with a power of 4.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

`standard_deviation`

Possible use:

- `standard_deviation` (`container`) —> `float`

Result:

the standard deviation on the elements of the operand. See Standard_deviation for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- standard_deviation ([4.5, 3.5, 5.5, 7.0]);    // 
   var0 equals 1.2930100540985752
```

See also:

mean, mean_deviation,

---

string

Possible use:

- string (any) —> string

Result:

Casts the operand into the type string

---

subtract_days

Possible use:

- date subtract_days int —> date
- subtract_days (date , int) —> date

Result:

Subtract a given number of days from a date

Examples:

```
date1 subtract_days 20
```

---

subtract_hours

Possible use:

- date subtract_hours int —> date
- subtract_hours (date , int) —> date

Result:

Add a given number of hours from a date

Examples:

```
date1 subtract_hours 15
```

---

subtract_minutes

Possible use:

- date subtract_minutes int —> date
- subtract_minutes (date , int) —> date

Result:

Subtract a given number of minutes from a date

Examples:

```
date1 subtract_minutes 5
```

---

subtract_months

Possible use:

- date subtract_months int —> date
- subtract_months (date , int) —> date

Result:

Subtract a given number of months from a date

Examples:

```
date1 subtract_months 5
```

---

subtract_seconds

Same signification as -

---

subtract_weeks

Possible use:

- date subtract_weeks int —> date
- subtract_weeks (date , int) —> date

Result:

Subtract a given number of weeks from a date

Examples:

```
date1 subtract_weeks 15
```

---

subtract_years

Possible use:

- date subtract_years int —> date
- subtract_years (date , int) —> date

Result:

Subtract a given number of year from a date

Examples:

```
date1 subtract_years 3
```

---

successors_of

Possible use:

- graph successors_of unknown —> container
- successors_of (graph , unknown) —> container

Result:

returns the list of successors (i.e. targets of out edges) of the given vertex (right-hand operand) in the given graph (left-hand operand)

Examples:

```
container var1 <- graphEpidemio successors_of ({1,5});  // var1
   equals [{12,45}]
container var2 <- graphEpidemio successors_of node({34,56});
   // var2 equals []
```

See also:

predecessors_of, neighbors_of,

`sum`

Possible use:

- `sum` (`graph`) —> `float`
- `sum` (`container`) —> `unknown`

Result:

the sum of all the elements of the operand

Comment:

the behavior depends on the nature of the operand

Special cases:

- if it is a population or a list of other types: sum transforms all elements into float and sums them
- if it is a map, sum returns the sum of the value of all elements
- if it is a file, sum returns the sum of the content of the file (that is also a container)
- if it is a graph, sum returns the sum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, sum returns the sum of all the numerical elements (i.e. all elements for integer and float matrices)

- if it is a matrix of geometry, sum returns the sum of the list of the geometries
- if it is a matrix of other types: sum transforms all elements into float and sums them
- if it is a list of colors: sum will sum them and return the blended resulting color
- if it is a list of int or float: sum returns the sum of all the elements

```
int var0 <- sum ([12,10,3]);     // var0 equals 25
```

- if it is a list of points: sum returns the sum of all points as a point (each coordinate is the sum of the corresponding coordinate of each element)

```
unknown var1 <- sum([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]);
      // var1 equals {20.0,17.0}
```

See also:

mul,

---

sum_of

Possible use:

- `container` `sum_of` `any expression` —> `unknown`
- `sum_of` (`container` , `any expression`) —> `unknown`

Result:

the sum of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var2 <- [1::2, 3::4, 5::6] sum_of (each + 3);   // var2
   equals 21
```

Examples:

```
unknown var1 <- [1,2] sum_of (each * 100 );     // var1 equals
   300
```

See also:

min_of, max_of, product_of, mean_of,

---

svg_file

Possible use:

- `svg_file` (`string`) —> `file`

Result:

Constructs a file of type svg. Allowed extensions are limited to svg

---

tan

Possible use:

- `tan` (`int`) —> `float`
- `tan` (`float`) —> `float`

Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized. Notice that tan(360) does not return 0.0 but -2.4492935982947064E-16

- The tangent is only defined for any real number except 90 + k * 180 (k an positive or negative integer). Nevertheless notice that tan(90) returns 1.633123935319537E16 (whereas we could except infinity).

Examples:

```
float var0 <- tan (0);  // var0 equals 0.0
float var1 <- tan(90);  // var1 equals 1.633123935319537E16
```

See also:

cos, sin,

`tan_rad`

Possible use:

- `tan_rad` (`float`) —> `float`

Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized. Notice that tan(360) does not return 0.0 but -2.4492935982947064E-16

- The tangent is only defined for any real number except 90 + k * 180 (k an positive or negative integer). Nevertheless notice that tan(90) returns 1.633123935319537E16 (whereas we could except infinity).

See also:

cos, sin,

`tanh`

Possible use:

- `tanh` (`float`) —> `float`
- `tanh` (`int`) —> `float`

Result:

Returns the value (in the interval [-1,1]) of the hyperbolic tangent of the operand (which can be any real number, expressed in decimal degrees).

Examples:

```
float var0 <- tanh(0);  // var0 equals 0.0
float var1 <- tanh(100);    // var1 equals 1.0
```

---

target_of

Possible use:

- graph target_of unknown —> unknown
- target_of (graph , unknown) —> unknown

Result:

returns the target of the edge (right-hand operand) contained in the graph given in left-hand operand.

Special cases:

- if the lef-hand operand (the graph) is nil, returns nil

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"
   ::edge,"vertices_specy"::node,"size"::3,"m"::5] );
unknown var1 <- graphEpidemio source_of(edge(3));   // var1
   equals node1
graph graphFromMap <-  as_edge_graph
   ([{1,5}::{12,45},{12,45}::{34,56}]);
unknown var3 <- graphFromMap target_of(link({1,5}::{12,45}));
   // var3 equals {12,45}
```

See also:

source_of,

---

teapot

Possible use:

- teapot (float) —> geometry

Result:

A teapot geometry which radius is equal to the operand.

Comment:

the centre of the teapot is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- teapot(10);     // var0 equals a geometry as a
   circle of radius 10 but displays a teapot.
```

See also:

around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

`text_file`

Possible use:

- `text_file` (`string`) —> `file`

Result:

Constructs a file of type text. Allowed extensions are limited to txt, data, text

---

`TGauss`

Same signification as truncated_gauss

`threeds_file`

Possible use:

- `threeds_file` (`string`) —> `file`

Result:

Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

---

`to`

Same signification as range

---

to_GAMA_CRS

Possible use:

- to_GAMA_CRS (geometry) —> geometry
- geometry to_GAMA_CRS string —> geometry
- to_GAMA_CRS (geometry , string) —> geometry

Special cases:

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by the current CRS, the one corresponding to the world's agent one

```
geometry var0 <- to_GAMA_CRS({121,14});     // var0 equals a
   geometry corresponding to the agent geometry transformed into
   the GAMA CRS
```

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by given CRS

```
geometry var1 <- to_GAMA_CRS({121,14}, "EPSG:4326");    // var1
   equals a geometry corresponding to the agent geometry
   transformed into the GAMA CRS
```

---

to_gaml

Possible use:

- to_gaml (unknown) —> string

Result:

returns the literal description of an expression or description – action, behavior, species, aspect, even model – in gaml

Examples:

```
string var0 <- to_gaml(0);  // var0 equals '0'
string var1 <- to_gaml(3.78);   // var1 equals '3.78'
string var2 <- to_gaml(true);   // var2 equals 'true'
string var3 <- to_gaml({23, 4.0});  // var3 equals
   '{23.0,4.0,0.0}'
string var4 <- to_gaml(5::34);  // var4 equals '5::34'
string var5 <- to_gaml(rgb(255,0,125));     // var5 equals 'rgb
   (255, 0, 125,255)'
string var6 <- to_gaml('hello');    // var6 equals "'hello'"
string var7 <- to_gaml([1,5,9,3]);  // var7 equals '[1,5,9,3]'
string var8 <- to_gaml(['a'::345, 'b'::13, 'c'::12]);   // var8
   equals "(['a'::345,'b'::13,'c'::12] as map )"
string var9 <- to_gaml([[3,5,7,9],[2,4,6,8]]);  // var9 equals
   '[[3,5,7,9],[2,4,6,8]]'
string var10 <- to_gaml(a_graph);   // var10 equals ([((1 as node
   )::(3 as node))::(5 as edge),((0 as node)::(3 as node))::(3 as
    edge),((1 as node)::(2 as node))::(1 as edge),((0 as node)
   ::(2 as node))::(2 as edge),((0 as node)::(1 as node))::(0 as
   edge),((2 as node)::(3 as node))::(4 as edge)] as map ) as
   graph
string var11 <- to_gaml(node1);     // var11 equals  1 as node
```

---

to_rectangles

Same signification as split_geometry

Possible use:

- to_rectangles (geometry, point, bool) —> list<geometry>

- `to_rectangles` (`geometry`, `int`, `int`, `bool`) —> `list<geometry>`

Result:

A list of rectangles of the size corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, dimension, overlaps), if overlaps = true, add the rectangles that overlap the border of the geometry A list of rectangles corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, nb_cols, nb_rows, overlaps) by a grid composed of the given number of columns and rows, if overlaps = true, add the rectangles that overlap the border of the geometry

Examples:

```
list<geometry> var0 <- to_rectangles(self, {10.0, 15.0}, true);
      // var0 equals the list of rectangles of size {10.0, 15.0}
   corresponding to the discretization into rectangles of the
   geometry of the agent applying the operator. The rectangles
   overlapping the border of the geometry are kept
list<geometry> var1 <- to_rectangles(self, 5, 20, true);    //
   var1 equals the list of rectangles corresponding to the
   discretization by a grid of 5 columns and 20 rows into
   rectangles of the geometry of the agent applying the operator.
    The rectangles overlapping the border of the geometry are
   kept
```

`to_squares`

Possible use:

- `to_squares` (`geometry`, `float`, `bool`) —> `list<geometry>`
- `to_squares` (`geometry`, `int`, `bool`) —> `list<geometry>`
- `to_squares` (`geometry`, `int`, `bool`, `float`) —> `list<geometry>`

Result:

A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps, precision_coefficient), if overlaps = true, add the squares that overlap the border of the geometry, coefficient_precision should be close to 1.0 A list of squares of the size corresponding to the given size that result from the decomposition of the geometry into squares (geometry, size, overlaps), if overlaps = true, add the squares that overlap the border of the geometry A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps), if overlaps = true, add the squares that overlap the border of the geometry

Examples:

```
list<geometry> var0 <- to_squares(self, 10, true, 0.99);    //
   var0 equals the list of 10 squares corresponding to the
   discretization into squares of the geometry of the agent
   applying the operator. The squares overlapping the border of
   the geometry are kept
list<geometry> var1 <- to_squares(self, 10.0, true);    // var1
   equals the list of squares of side size 10.0 corresponding to
   the discretization into squares of the geometry of the agent
   applying the operator. The squares overlapping the border of
   the geometry are kept
list<geometry> var2 <- to_squares(self, 10, true);  // var2
   equals the list of 10 squares corresponding to the
   discretization into squares of the geometry of the agent
   applying the operator. The squares overlapping the border of
   the geometry are kept
```

to_triangles

Same signification as triangulate

`tokenize`

Same signification as split_with

`topology`

Possible use:

- `topology` (`unknown`) —> `topology`

Result:

casting of the operand to a topology.

Special cases:

- if the operand is a topology, returns the topology itself;
- if the operand is a spatial graph, returns the graph topology associated;
- if the operand is a population, returns the topology of the population;
- if the operand is a shape or a geometry, returns the continuous topology bounded by the geometry;
- if the operand is a matrix, returns the grid topology associated
- if the operand is another kind of container, returns the multiple topology associated to the container
- otherwise, casts the operand to a geometry and build a topology from it.

Examples:

```
topology var0 <- topology(0);    // var0 equals nil
topology(a_graph)   --: Multiple topology in POLYGON
   ((24.712119771887785 7.867357373616512, 24.712119771887785
   61.283226839310565, 82.4013676510046  7.867357373616512)) at
   location[53.556743711446195;34.57529210646354]
```

See also:

geometry,

---

touches

Possible use:

- `geometry` `touches` `geometry` —> `bool`
- `touches` (`geometry` , `geometry`) —> `bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) touches the right-geometry (or agent/point).

Comment:

returns true when the left-operand only touches the right-operand. When one geometry covers partially (or fully) the other one, it returns false.

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) touches {15,15};   //
   var0 equals false
bool var1 <- polyline([{10,10},{20,20}]) touches {10,10};   //
   var1 equals true
bool var2 <- {15,15} touches {15,15};   // var2 equals false
bool var3 <- polyline([{10,10},{20,20}]) touches polyline
   ([{10,10},{5,5}]);     // var3 equals true
bool var4 <- polyline([{10,10},{20,20}]) touches polyline
   ([{5,5},{15,15}]);     // var4 equals false
bool var5 <- polyline([{10,10},{20,20}]) touches polyline
   ([{15,15},{25,25}]);   // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
   polygon([{15,15},{15,25},{25,25},{25,15}]);     // var6 equals
    false
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
   polygon([{10,20},{20,20},{20,30},{10,30}]);     // var7 equals
    true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
   polygon([{10,10},{0,10},{0,0},{10,0}]);     // var8 equals
   true
bool var9 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
   {15,15};    // var9 equals false
bool var10 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
   {10,15};   // var10 equals true
```

See also:

disjoint_from, crosses, overlaps, partially_overlaps, intersects,

---

`towards`

Possible use:

- `geometry` **`towards`** `geometry` —> `int`
- **`towards`** (`geometry` , `geometry`) —> `int`

Result:

The direction (in degree) between the two geometries (geometries, agents, points) considering the topology of the agent applying the operator.

Examples:

```
int var0 <- ag1 towards ag2;     // var0 equals the direction
   between ag1 and ag2 and ag3 considering the topology of the
   agent applying the operator
```

See also:

distance_between, distance_to, direction_between, path_between, path_to,

---

trace

Possible use:

- trace (matrix) —> float

Result:

The trace of the given matrix (the sum of the elements on the main diagonal).

Examples:

```
float var0 <- trace(matrix([[1,2],[3,4]]));     // var0 equals 5
```

`transformed_by`

Possible use:

- `geometry` `transformed_by` `point` —> `geometry`
- `transformed_by` (`geometry` , `point`) —> `geometry`

Result:

A geometry resulting from the application of a rotation and a scaling (right-operand : point {angle(degree), scale factor} of the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self transformed_by {45, 0.5};     // var0
   equals the geometry resulting from 45 degrees rotation and 50%
    scaling of the geometry of the agent applying the operator.
```

See also:

rotated_by, translated_by,

`translated_by`

Possible use:

- `geometry` `translated_by` `point` —> `geometry`
- `translated_by` (`geometry` , `point`) —> `geometry`

Result:

A geometry resulting from the application of a translation by the right-hand operand distance to the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self translated_by {10,10,10};      // var0
   equals the geometry resulting from applying the translation to
    the left-hand geometry (or agent).
```

See also:

rotated_by, transformed_by,

---

translated_to

Same signification as at_location

---

transpose

Possible use:

- `transpose` (`matrix`) —> `matrix`

Result:

The transposition of the given matrix

Examples:

```
matrix var0 <- transpose(matrix([[5,-3],[6,-4]]));  // var0
   equals [[5,6],[-3,-4]]
```

`triangle`

Possible use:

- `triangle` (`float`) —> `geometry`

Result:

A triangle geometry which side size is given by the operand.

Comment:

the center of the triangle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- triangle(5);   // var0 equals a geometry as a
   triangle with side_size = 5.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, square,

---

`triangulate`

Possible use:

- `triangulate` (`list<geometry>`) —> `list<geometry>`
- `triangulate` (`geometry`) —> `list<geometry>`

Result:

A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point)

Examples:

```
list<geometry> var0 <- triangulate(self);   // var0 equals the
   list of geometries (triangles) corresponding to the Delaunay
   triangulation of the geometry of the agent applying the
   operator.
list<geometry> var1 <- triangulate(self);   // var1 equals the
   list of geometries (triangles) corresponding to the Delaunay
   triangulation of the geometry of the agent applying the
   operator.
```

`truncated_gauss`

Possible use:

- `truncated_gauss` (`point`) —> `float`
- `truncated_gauss` (`container`) —> `float`

Result:

A random value from a normally distributed random variable in the interval]mean - standardDeviation; mean + standardDeviation[.

Special cases:

- when the operand is a point, it is read as {mean, standardDeviation}

- if the operand is a list, only the two first elements are taken into account as [mean, standardDeviation]
- when truncated_gauss is called with a list of only one element mean, it will always return 0.0

Examples:

```
float var0 <- truncated_gauss ({0, 0.3});   // var0 equals a
   float between -0.3 and 0.3
float var1 <- truncated_gauss ([0.5, 0.0]);     // var1 equals
   0.5
```

See also:

gauss,

---

undirected

Possible use:

- `undirected` (`graph`) —> `graph`

Result:

the operand graph becomes an undirected graph.

Comment:

the operator alters the operand graph, it does not create a new one.

See also:

directed,

---

union

Same signification as +

Possible use:

- `union` (`container<geometry>`) —> `geometry`
- `container` `union` `container` —> `container`
- `union` (`container` , `container`) —> `container`

Result:

returns a new list containing all the elements of both containers without duplicated elements.

Special cases:

- if the right-operand is a container of points, geometries or agents, returns the geometry resulting from the union all the geometries

- if the left or right operand is nil, union throws an error

Examples:

```
geometry var0 <- union([geom1, geom2, geom3]);  // var0 equals a
   geometry corresponding to union between geom1, geom2 and geom3
container var1 <- [1,2,3,4,5,6] union [2,4,9];  // var1 equals
   [1,2,3,4,5,6,9]
container var2 <- [1,2,3,4,5,6] union [0,8];    // var2 equals
   [1,2,3,4,5,6,0,8]
```

```
container var3 <- [1,3,2,4,5,6,8,5,6] union [0,8];  // var3
   equals [1,3,2,4,5,6,8,0]
```

See also:

inter, +,

---

unknown

Possible use:

- `unknown` (any) —> `unknown`

Result:

Casts the operand into the type unknown

---

upper_case

Possible use:

- `upper_case` (`string`) —> `string`

Result:

Converts all of the characters in the string operand to upper case

Examples:

```
string var0 <- upper_case("Abc");   // var0 equals 'ABC'
```

See also:

lower_case,

`URL_file`

Possible use:

- `URL_file` (`string`) —> `file`

Result:

Constructs a file of type URL. Allowed extensions are limited to url

---

`use_cache`

Possible use:

- `graph` `use_cache` `bool` —> `graph`
- `use_cache` (`graph` , `bool`) —> `graph`

Result:

if the second operand is true, the operand graph will store in a cache all the previously computed shortest path (the cache be cleared if the graph is modified).

Comment:

the operator alters the operand graph, it does not create a new one.

See also:

path_between,

---

user_input

Possible use:

- `user_input (any expression) —> map<string,unknown>`
- `string user_input any expression —> map<string,unknown>`
- `user_input (string , any expression) —> map<string,unknown>`

Result:

asks the user for some values (not defined as parameters). Takes a string (optional) and a map as arguments. The string is used to specify the message of the dialog box. The map is to specify the parameters you want the user to change before the simulation starts, with the name of the parameter in string key, and the default value as value.

Comment:

This operator takes a map [string::value] as argument, displays a dialog asking the user for these values, and returns the same map with the modified values (if any). The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section to force the user to input new values instead of relying on the initial values of parameters :

Examples:

```
map<string,unknown> values <- user_input(["Number" :: 100, "
    Location" :: {10, 10}]);
create bug number: int(values at "Number") with: [location:: (
    point(values at "Location"))];
```

```
map<string,unknown> values2 <- user_input("Enter numer of agents
    and locations",["Number" :: 100, "Location" :: {10, 10}]);
create bug number: int(values2 at "Number") with: [location:: (
    point(values2 at "Location"))];
```

---

using

Possible use:

- any expression using topology —> unknown
- using (any expression , topology) —> unknown

Result:

Allows to specify in which topology a spatial computation should take place.

Special cases:

- has no effect if the topology passed as a parameter is nil

Examples:

```
unknown var0 <- (agents closest_to self) using topology(world);
      // var0 equals the closest agent to self (the caller) in
    the continuous topology of the world
```

---

variance

Possible use:

- variance (container) —> float

Result:

the variance of the elements of the operand. See Variance for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- variance ([4.5, 3.5, 5.5, 7.0]);  // var0 equals
   1.671875
```

See also:

mean, median,

---

variance_of

Possible use:

- `container` `variance_of` `any expression` —> `unknown`
- `variance_of` (`container` , `any expression`) —> `unknown`

Result:

the variance of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

See also:

min_of, max_of, sum_of, product_of,

---

voronoi

Possible use:

- `voronoi` (`list<point>`) —> `list<geometry>`
- `list<point>` `voronoi` `geometry` —> `list<geometry>`
- `voronoi` (`list<point>` , `geometry`) —> `list<geometry>`

Result:

A list of geometries corresponding to the Voronoi diagram built from the list of points A list of geometries corresponding to the Voronoi diagram built from the list of points according to the given clip

Examples:

```
list<geometry> var0 <- voronoi
   ([{10,10},{50,50},{90,90},{10,90},{90,10}]);  // var0 equals
   the list of geometries corresponding to the Voronoi Diagram
   built from the list of points.
list<geometry> var1 <- voronoi
   ([{10,10},{50,50},{90,90},{10,90},{90,10}], square(300));
   // var1 equals the list of geometries corresponding to the
   Voronoi Diagram built from the list of points with a square of
    300m side size as clip.
```

---

weight_of

Possible use:

- graph weight_of unknown —> float
- weight_of (graph , unknown) —> float

Result:

returns the weight of the given edge (right-hand operand) contained in the graph given in right-hand operand.

Comment:

In a localized graph, an edge has a weight by default (the distance between both vertices).

Special cases:

- if the left-operand (the graph) is nil, returns nil
- if the right-hand operand is not an edge of the given graph, weight_of checks whether it is a node of the graph and tries to return its weight
- if the right-hand operand is neither a node, nor an edge, returns 1.

Examples:

```
graph graphFromMap <-  as_edge_graph
   ([{1,5}::{12,45},{12,45}::{34,56}]);
float var1 <- graphFromMap weight_of(link({1,5}::{12,45}));
   // var1 equals 1.0
```

weighted_means_DM

Possible use:

- list<list> weighted_means_DM list<map<string,object>> —> int
- weighted_means_DM (list<list> , list<map<string,object>>) —> int

Result:

The index of the candidate that maximizes the weighted mean of its criterion values. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion (list of map)

Special cases:

- returns -1 is the list of candidates is nil or empty

Examples:

```
int var0 <- weighted_means_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]],
    [["name"::"utility", "weight" :: 2.0],["name"::"price", "
    weight" :: 1.0]]);     // var0 equals 1
```

See also:

promethee_DM, electre_DM, evidence_theory_DM,

---

where

Possible use:

- container where any expression —> container
- where (container , any expression) —> container

Result:

a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is a list nil, where returns a new empty list
- if the left-operand is a map, the keyword each will contain each value

```
container var4 <- [1::2, 3::4, 5::6] where (each >= 4);     //
   var4 equals [4, 6]
```

Examples:

```
container var0 <- [1,2,3,4,5,6,7,8] where (each > 3);   // var0
   equals [4, 5, 6, 7, 8]
container var2 <- g2 where (length(g2 out_edges_of each) = 0 );
      // var2 equals [node9, node7, node10, node8, node11]
container var3 <- (list(node) where (round(node(each).location.x)
    > 32);    // var3 equals [node2, node3]
```

See also:

first_with, last_with, where,

`with_lifetime`

Possible use:

- `predicate` `with_lifetime` `int` —> `predicate`
- `with_lifetime` (`predicate` , `int`) —> `predicate`

Result:

change the parameters of the given predicate

Examples:

```
predicate with_lifetime 10
```

`with_max_of`

Possible use:

- `container` `with_max_of` `any expression` —> `unknown`
- `with_max_of` (`container` , `any expression`) —> `unknown`

Result:

one of elements of the left-hand operand that maximizes the value of the right-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_max_of (each );  // var0
   equals 8
unknown var2 <- g2 with_max_of (length(g2 out_edges_of each)  ) ;
      // var2 equals node4
unknown var3 <- (list(node) with_max_of (round(node(each).
   location.x));      // var3 equals node3
unknown var4 <- [1::2, 3::4, 5::6] with_max_of (each);  // var4
   equals 6
```

See also:

where, with_min_of,

`with_min_of`

Possible use:

- `container` `with_min_of` `any expression` —> `unknown`
- `with_min_of` (`container` , `any expression`) —> `unknown`

Result:

one of elements of the left-hand operand that minimizes the value of the right-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_min_of (each );  // var0
   equals 1
unknown var2 <- g2 with_min_of (length(g2 out_edges_of each)  );
      // var2 equals node11
unknown var3 <- (list(node) with_min_of (round(node(each).
   location.x));      // var3 equals node0
unknown var4 <- [1::2, 3::4, 5::6] with_min_of (each);  // var4
   equals 2
```

See also:

where, with_max_of,

---

with_optimizer_type

Possible use:

- graph `with_optimizer_type` string —> graph
- `with_optimizer_type` (graph , string) —> graph

Result:

changes the shortest path computation method of the given graph

Comment:

the right-hand operand can be “Djikstra”, “Bellmann”, “Astar” to use the associated algorithm. Note that these methods are dynamic: the path is computed when needed. In contrarily, if the operand is another string, a static method will be used, i.e. all the shortest are previously computed.

Examples:

```
graphEpidemio <- graphEpidemio with_optimizer_type "static";
```

See also:

set_verbose,

`with_precision`

Possible use:

- `point` `with_precision` `int` —> `point`
- `with_precision` (`point` , `int`) —> `point`
- `float` `with_precision` `int` —> `float`
- `with_precision` (`float` , `int`) —> `float`

Result:

Rounds off the ordinates of the left-hand point to the precision given by the value of right-hand operand Rounds off the value of left-hand operand to the precision given by the value of right-hand operand

Examples:

```
point var0 <- {12345.78943, 12345.78943, 12345.78943}
   with_precision 2 ;    // var0 equals {12345.79, 12345.79,
   12345.79}
float var1 <- 12345.78943 with_precision 2;     // var1 equals
   12345.79
float var2 <- 123 with_precision 2;     // var2 equals 123.00
```

See also:

round,

---

with_priority

Possible use:

- `predicate` `with_priority` `float` —> `predicate`
- `with_priority` (`predicate` , `float`) —> `predicate`

Result:

change the priority of the given predicate

Examples:

```
predicate with_priority 2
```

---

with_values

Possible use:

- predicate with_values map —> predicate
- with_values (predicate , map) —> predicate

Result:

change the parameters of the given predicate

Examples:

```
predicate with_values ["time"::10]
```

---

with_weights

Possible use:

- graph with_weights container —> graph
- with_weights (graph , container) —> graph
- graph with_weights map —> graph
- with_weights (graph , map) —> graph

Result:

returns the graph (left-hand operand) with weight given in the map (right-hand operand).

Comment:

this operand re-initializes the path finder

Special cases:

- if the right-hand operand is a list, affects the n elements of the list to the n first edges. Note that the ordering of edges may change overtime, which can create some problems...
- if the left-hand operand is a map, the map should contains pairs such as: vertex/edge::double

```
graph_from_edges (list(ant) as_map each::one_of (list(ant)))
   with_weights (list(ant) as_map each::each.food)
```

---

without_holes

Possible use:

- `without_holes` (`geometry`) —> `geometry`

Result:

A geometry corresponding to the operand geometry (geometry, agent, point) without its holes

Examples:

```
geometry var0 <- solid(self);   // var0 equals the geometry
   corresponding to the geometry of the agent applying the
   operator without its holes.
```

`writable`

Possible use:

- `file` `writable` `bool` —> `file`
- `writable` (`file` , `bool`) —> `file`

Result:

Marks the file as read-only or not, depending on the second boolean argument, and returns the first argument

Comment:

A file is created using its native flags. This operator can change them. Beware that this change is system-wide (and not only restrained to GAMA): changing a file to read-only mode (e.g. "writable(f, false)")

Examples:

```
file var0 <- shape_file("../images/point_eau.shp") writable false
    ;  // var0 equals returns a file in read-only mode
```

See also:

file,

`xml_file`

Possible use:

- `xml_file` (`string`) —> `file`

Result:

Constructs a file of type xml. Allowed extensions are limited to xml

References

This page contains a subset of the scientific papers that have been written either about GAMA or using the platform as an experimental/modeling support.

If you happen to publish a paper that uses or discusses GAMA, please let us know, so that we can include it in this list.

As stated in the first page, if you need to cite GAMA in a paper, we kindly ask you to use this reference: * A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

Papers about GAMA

- Taillandier, Patrick, Arnaud Grignard, Benoit Gaudou, and Alexis Drogoul. "Des données géographiques à la simulation à base d'agents: application de la plate-forme GAMA." Cybergeo: European Journal of Geography (2014).

- A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

- Grignard, Arnaud, Alexis Drogoul, and Jean-Daniel Zucker. "Online analysis and visualization of agent based models." Computational Science and Its Applications–ICCSA 2013. Springer Berlin Heidelberg, 2013. 662-672.

- Taillandier, P., Drogoul, A., Vo, D.A. and Amouroux, E. (2012), GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp 242-258.

- Taillandier, P. & Drogoul, A. (2011), From Grid Environment to Geographic Vector Agents, Modeling with the GAMA simulation platform. In '25th Conference of the International Cartographic Association', Paris, France.

- Taillandier, P. ; Drogoul A. ; Vo D.A. & Amouroux, E. (2010), GAMA : bringing GIS and multi-level capabilities to multi-agent simulation, in 'the 8th European Workshop on Multi-Agent Systems', Paris, France.

- Amouroux, E., Taillandier, P. & Drogoul, A. (2010), Complex environment representation in epidemiology ABM: application on `H5N1` propagation. In 'the 3rd International Conference on Theories and Applications of Computer Science' (ICTACS'10).

- Amouroux, E., Chu, T.Q., Boucher, A. and Drogoul, A. (2007), GAMA: an environment for implementing and running spatially explicit multi-agent simulations. In 'Pacific Rim International Workshop on Multi-Agents', Bangkoku, Thailand, pp. 359–371.

PhD theses

- **Truong Xuan Viet**, "Optimization by Simulation of an Environmental Surveillance Network: Application to the Fight against Rice Pests in the Mekong Delta (Vietnam)", University of Paris 6 & Ho Chi Minh University of Technology, defended June 24th, 2014.

- **Nguyen Nhi Gia Vinh**, "Designing multi-scale models to support environmental decision: application to the control of Brown Plant Hopper invasions in the Mekong Delta (Vietnam)", University of Paris 6, defended Oct. 31st, 2013.

- **Vo Duc An**, "An operational architecture to handle multiple levels of representation in agent-based models", University of Paris 6, defended Nov. 30th 2012.

- **Amouroux Edouard**, "KIMONO: a descriptive agent-based modeling methodology for the exploration of complex systems: an application to epidemiology", University of Paris 6, defended Sept. 30th, 2011.

- **Chu Thanh Quang**, "Using agent-based models and machine learning to enhance spatial decision support systems: Application to resource allocation in situations of urban catastrophes", University of Paris 6, defended July 1st, 2011.

- **Nguyen Ngoc Doanh**, "Coupling Equation-Based and Individual-Based Models in the Study of Complex Systems: A Case Study in Theoretical Population Ecology", University of Paris 6, defended Dec. 14th, 2010.

Research papers that use GAMA as modeling/simulation support

2016

- Bhamidipati, S., van der Lei, T., & Herder, P. (2016). A layered approach to model interconnected infrastructure and its significance for asset management. EJTIR, 16(1), 254-272.

2014

- E. G. Macatulad , A. C. Blanco (2014) 3DGIS-BASED MULTI-AGENT GEOSIMULATION AND VISUALIZATION OF BUILDING EVACUATION USING GAMA PLATFORM. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-2, 2014. ISPRS Technical Commission II Symposium, 6 – 8 October 2014, Toronto, Canada. Retrieved from http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-2/87/2014/isprsarchives-XL-2-87-2014.pdf

- S. Bhamidipati (2014) A simulation framework for asset management in climate-change adaptation of transportation infrastructure. In: Proceedings of 42nd European Transport Conference. Frankfurt, Germany. Retrieved from http://abstracts.aetransport.org/paper/download/id/4317

- Gaudou, B., Sibertin-Blanc, C., Thérond, O., Amblard, F., Auda, Y., Arcangeli, J.-P., Balestrat, M., Charron-Moirez, M.-H., Gondet, E., Hong, Y., Lardy, R., Louail, T., Mayor, E., Panzoli, D., Sauvage, S., Sanchez-Perez, J., Taillandier, P., Nguyen, V. B., Vavasseur, M., Mazzega, P. (2014). The MAELIA multi-agent platform for integrated assessment of low-water management issues. In: International Workshop on Multi-Agent-Based Simulation (MABS 2013), Saint-Paul, MN, USA, 06/05/2013-07/05/2013, Vol. 8235, Shah Jamal Alam, H. Van Dyke Parunak, (Eds.), Springer, Lecture Notes in Computer Science, p. 85-110.

- Gaudou, B., Lorini, E., Mayor, E. (2014.) Moral Guilt: An Agent-Based Model Analysis. In: Conference of the European Social Simulation Association (ESSA 2013), Warsaw, 16/09/2013-20/09/2013, Vol. 229, Springer, Advances in Intelligent Systems and Computing, p. 95-106.

2013 * Drogoul, A., Gaudou, B., Grignard, A., Taillandier, P., & Vo, D. A. (2013). Practical Approach To Agent-Based Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, p. 277-300, Regional Social Sciences Summer University.

- Drogoul, A., Gaudou, B. (2013) Methods for Agent-Based Computer Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, 1.6, p. 130-154, Regional Social Sciences Summer University.

- Truong, M.-T., Amblard, F., Gaudou, B., Sibertin-Blanc, C., Truong, V. X., Drogoul, A., Hyunh, X. H., Le, M. N. (2013). An implementation of framework of business intelligence for agent-based simulation. In: Symposium on Information and Communication Technology (SoICT 2013), Da Nang, Viet Nam, 05/12/2013-06/12/2013, Quyet Thang Huynh, Thanh Binh Nguyen, Van Tien Do, Marc Bui, Hong Son Ngo (Eds.), ACM, p. 35-44.

- Le, V. M., Gaudou, B., Taillandier, P., Vo, D. A (2013). A New BDI Architecture To Formalize Cognitive Agent Behaviors Into Simulations. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems (KES-AMSTA 2013), Hue, Vietnam, 27/05/2013-29/05/2013, Vol. 252, Dariusz Barbucha, Manh Thanh Le, Robert J. Howlett, C. Jain Lakhmi (Eds.), IOS Press, Frontiers in Artificial Intelligence and Applications, p. 395-403.

2012 * Taillandier, P., Therond, O., Gaudou B. (2012), A new BDI agent architecture based on the belief theory. Application to the modelling of cropping plan decision-making. In 'International Environmental Modelling and Software Society', Germany, pp. 107-116.

- Taillandier, P., Therond, O., Gaudou B. (2012), Une architecture d'agent BDI basée sur la théorie des fonctions de croyance: application à la simulation du comportement des agriculteurs. In 'Journées Francophones sur les Systèmes Multi-Agents', France, pp. 107-116.

- NGUYEN, Quoc Tuan, Alain BOUJU, and Pascal ESTRAILLIER. "Multi-agent architecture with space-time components for the simulation of urban transportation systems." (2012).
- Cisse, A., Bah, A., Drogoul, A., Cisse, A.T., Ndione, J.A., Kebe, C.M.F. & Taillandier P. (2012), Un modèle à base d'agents sur la transmission et la diffusion de la fièvre de la Vallée du Rift à Barkédji (Ferlo, Sénégal), Studia Informatica Universalis 10 (1), pp. 77-97.
- Taillandier, P., Amouroux, E., Vo, D.A. and Olteanu-Raimond A.M. (2012), Using Belief Theory to formalize the agent behavior: application to the simulation of avian flu propagation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 575-587.
- Le, V.M., Adam, C., Canal, R., Gaudou, B., Ho, T.V. and Taillandier, P. (2012), Simulation of the emotion dynamics in a group of agents in an evacuation situation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 604-619.
- Nguyen Vu, Q. A., Canal, R., Gaudou, B., Hassas, S., Armetta, F. (2012), TrustSets - Using trust to detect deceitful agents in a distributed information collecting system. In: Journal of Ambient Intelligence and Humanized Computing, Springer-Verlag, Vol. 3 N. 4, p. 251-263.

2011 * Taillandier, P. & Therond, O. (2011), Use of the Belief Theory to formalize Agent Decision Making Processes : Application to cropping Plan Decision Making. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 138-142.

- Taillandier, P. & Amblard, F. (2011), Cartography of Multi-Agent Model Parameter Space through a reactive Dicotomous Approach. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 38-42.
- Taillandier, P. & Stinckwich, S. (2011), Using the PROMETHEE Multi-Criteria Decision Making Method to Define New Exploration Strategies for Rescue Robots', IEEE International Symposium on Safety, Security, and Rescue Robotics, Kyoto, Japon, pp. 321 - 326.

2010 * Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), A cluster-based approach for disturbed, spatialized, distributed information gathering systems, in 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, pp. 588-603.

- Nguyen, N.D., Taillandier, P., Drogoul, A. and Augier, P. (2010), Inferring Equation-Based Models from Agent-Based Models: A Case Study in Competition Dynamics.In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp. 413-427.
- Amouroux, E., Gaudou, B. Desvaux, S. and Drogoul, A. (2010), O.D.D.: a Promising but Incomplete Formalism For Individual-Based Model Specification. in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), pp. 1-4.
- Nguyen, N.D., Phan, T.H.D., Nguyen, T.N.A., Drogoul, A. and Zucker, J-D. (2010), Disk Graph-Based Model for Competition Dynamic, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen, T.K., Marilleau, N., Ho T.V. and El Fallah Seghrouchni, A. (2010), A meta-model for specifying collaborative simulation, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), `TrustSets` - Using trust to detect deceitful agents in a distributed information collecting system, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), the best student paper award.
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S., Armetta, F. and Stinckwich, S. (2010), Using trust and cluster organisation to improve robot swarm mapping, Paper to appear in 'Workshop on Robots and Sensors integration in future rescue INFormation system' (ROSIN 2010).

2009 * Taillandier, P. and Buard, E. (2009), Designing Agent Behaviour in Agent-Based Simulation through participatory method. In 'The 12th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Nagoya, Japan, pp. 571–578.

- Taillandier, P. and Chu, T.Q. (2009), Using Participatory Paradigm to Learn Human Behaviour. In 'International Conference on Knowledge and Systems Engineering', Ha noi, Viet Nam, pp. 55–60.
- Gaudou, B., Ho, T.V. and Marilleau, N. (2009), Introduce collaboration in methodologies of modeling and simulation of Complex Systems. In 'International Conference on Intelligent Networking and Collaborative Systems (INCOS '09)'. Barcelona, pp. 1–8.

- Nguyen, T.K., Gaudou B., Ho T.V. and Marilleau N. (2009), Application of PAMS Collaboration Platform to Simulation-Based Researches in Soil Science: The Case of the MIcro-ORganism Project. In 'IEEE International Conference on Computing and Telecommunication Technologies (IEEE-RIVF 09)'. Da Nang, Viet Nam, pp. 296–303.

- Nguyen, V.Q., Gaudou B., Canal R., Hassas S. and Armetta F. (2009), Stratégie de communication dans un système de collecte d'information à base d'agents perturbés. In 'Journées Francophones sur les Systèmes Multi-Agents (JFSMA'09)'.

2008 * Chu, T.Q., Boucher, A., Drogoul, A., Vo, D.A., Nguyen, H.P. and Zucker, J.D. (2008). Interactive Learning of Expert Criteria for Rescue Simulations. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 127–138.

- Amouroux, E., Desvaux, S. and Drogoul, A. (2008), Towards Virtual Epidemiology: An Agent-Based Approach to the Modeling of H5N1 Propagation and Persistence in North-Vietnam. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 26–33.

References

This page contains a subset of the scientific papers that have been written either about GAMA or using the platform as an experimental/modeling support.

If you happen to publish a paper that uses or discusses GAMA, please let us know, so that we can include it in this list.

As stated in the first page, if you need to cite GAMA in a paper, we kindly ask you to use this reference: * A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

Papers about GAMA

- Taillandier, Patrick, Arnaud Grignard, Benoit Gaudou, and Alexis Drogoul. "Des données géographiques à la simulation à base d'agents: application de la plate-forme GAMA." Cybergeo: European Journal of Geography (2014).

- A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

- Grignard, Arnaud, Alexis Drogoul, and Jean-Daniel Zucker. "Online analysis and visualization of agent based models." Computational Science and Its Applications–ICCSA 2013. Springer Berlin Heidelberg, 2013. 662-672.

- Taillandier, P., Drogoul, A., Vo, D.A. and Amouroux, E. (2012), GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp 242-258.

- Taillandier, P. & Drogoul, A. (2011), From Grid Environment to Geographic Vector Agents, Modeling with the GAMA simulation platform. In '25th Conference of the International Cartographic Association', Paris, France.

- Taillandier, P. ; Drogoul A. ; Vo D.A. & Amouroux, E. (2010), GAMA : bringing GIS and multi-level capabilities to multi-agent simulation, in 'the 8th European Workshop on Multi-Agent Systems', Paris, France.

- Amouroux, E., Taillandier, P. & Drogoul, A. (2010), Complex environment representation in epidemiology ABM: application on `H5N1` propagation. In 'the 3rd International Conference on Theories and Applications of Computer Science' (ICTACS'10).

- Amouroux, E., Chu, T.Q., Boucher, A. and Drogoul, A. (2007), GAMA: an environment for implementing and running spatially explicit multi-agent simulations. In 'Pacific Rim International Workshop on Multi-Agents', Bangkoku, Thailand, pp. 359–371.

PhD theses

- **Truong Xuan Viet**, "Optimization by Simulation of an Environmental Surveillance Network: Application to the Fight against Rice Pests in the Mekong Delta (Vietnam)", University of Paris 6 & Ho Chi Minh University of Technology, defended June 24th, 2014.

- **Nguyen Nhi Gia Vinh**, "Designing multi-scale models to support environmental decision: application to the control of Brown Plant Hopper invasions in the Mekong Delta (Vietnam)", University of Paris 6, defended Oct. 31st, 2013.

- **Vo Duc An**, "An operational architecture to handle multiple levels of representation in agent-based models", University of Paris 6, defended Nov. 30th 2012.

- **Amouroux Edouard**, "KIMONO: a descriptive agent-based modeling methodology for the exploration of complex systems: an application to epidemiology", University of Paris 6, defended Sept. 30th, 2011.

- **Chu Thanh Quang**, "Using agent-based models and machine learning to enhance spatial decision support systems: Application to resource allocation in situations of urban catastrophes", University of Paris 6, defended July 1st, 2011.

- **Nguyen Ngoc Doanh**, "Coupling Equation-Based and Individual-Based Models in the Study of Complex Systems: A Case Study in Theoretical Population Ecology", University of Paris 6, defended Dec. 14th, 2010.

Research papers that use GAMA as modeling/simulation support

2016

- Bhamidipati, S., van der Lei, T., & Herder, P. (2016). A layered approach to model interconnected infrastructure and its significance for asset management. EJTIR, 16(1), 254-272.

2014

- E. G. Macatulad , A. C. Blanco (2014) 3DGIS-BASED MULTI-AGENT GEOSIMULATION AND VISUALIZATION OF BUILDING EVACUATION USING GAMA PLATFORM. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-2, 2014. ISPRS Technical Commission II Symposium, 6 – 8 October 2014, Toronto, Canada. Retrieved from http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-2/87/2014/isprsarchives-XL-2-87-2014.pdf

- S. Bhamidipati (2014) A simulation framework for asset management in climate-change adaptation of transportation infrastructure. In: Proceedings of 42nd European Transport Conference. Frankfurt, Germany. Retrieved from http://abstracts.aetransport.org/paper/download/id/4317

- Gaudou, B., Sibertin-Blanc, C., Thérond, O., Amblard, F., Auda, Y., Arcangeli, J.-P., Balestrat, M., Charron-Moirez, M.-H., Gondet, E., Hong, Y., Lardy, R., Louail, T., Mayor, E., Panzoli, D., Sauvage, S., Sanchez-Perez, J., Taillandier, P., Nguyen, V. B., Vavasseur, M., Mazzega, P. (2014). The MAELIA multi-agent platform for integrated assessment of low-water management issues. In: International Workshop on Multi-Agent-Based Simulation (MABS 2013), Saint-Paul, MN, USA, 06/05/2013-07/05/2013, Vol. 8235, Shah Jamal Alam, H. Van Dyke Parunak, (Eds.), Springer, Lecture Notes in Computer Science, p. 85-110.

- Gaudou, B., Lorini, E., Mayor, E. (2014.) Moral Guilt: An Agent-Based Model Analysis. In: Conference of the European Social Simulation Association (ESSA 2013), Warsaw, 16/09/2013-20/09/2013, Vol. 229, Springer, Advances in Intelligent Systems and Computing, p. 95-106.

2013 * Drogoul, A., Gaudou, B., Grignard, A., Taillandier, P., & Vo, D. A. (2013). Practical Approach To Agent-Based Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, p. 277-300, Regional Social Sciences Summer University.

- Drogoul, A., Gaudou, B. (2013) Methods for Agent-Based Computer Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, 1.6, p. 130-154, Regional Social Sciences Summer University.

- Truong, M.-T., Amblard, F., Gaudou, B., Sibertin-Blanc, C., Truong, V. X., Drogoul, A., Hyunh, X. H., Le, M. N. (2013). An implementation of framework of business intelligence for agent-based simulation. In: Symposium on Information and Communication Technology (SoICT 2013), Da Nang, Viet Nam, 05/12/2013-06/12/2013, Quyet Thang Huynh, Thanh Binh Nguyen, Van Tien Do, Marc Bui, Hong Son Ngo (Eds.), ACM, p. 35-44.

- Le, V. M., Gaudou, B., Taillandier, P., Vo, D. A (2013). A New BDI Architecture To Formalize Cognitive Agent Behaviors Into Simulations. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems (KES-AMSTA 2013), Hue, Vietnam, 27/05/2013-29/05/2013, Vol. 252, Dariusz Barbucha, Manh Thanh Le, Robert J. Howlett, C. Jain Lakhmi (Eds.), IOS Press, Frontiers in Artificial Intelligence and Applications, p. 395-403.

2012 * Taillandier, P., Therond, O., Gaudou B. (2012), A new BDI agent architecture based on the belief theory. Application to the modelling of cropping plan decision-making. In 'International Environmental Modelling and Software Society', Germany, pp. 107-116.

- Taillandier, P., Therond, O., Gaudou B. (2012), Une architecture d'agent BDI basée sur la théorie des fonctions de croyance: application à la simulation du comportement des agriculteurs. In 'Journées Francophones sur les Systèmes Multi-Agents', France, pp. 107-116.

- NGUYEN, Quoc Tuan, Alain BOUJU, and Pascal ESTRAILLIER. "Multi-agent architecture with space-time components for the simulation of urban transportation systems." (2012).
- Cisse, A., Bah, A., Drogoul, A., Cisse, A.T., Ndione, J.A., Kebe, C.M.F. & Taillandier P. (2012), Un modèle à base d'agents sur la transmission et la diffusion de la fièvre de la Vallée du Rift à Barkédji (Ferlo, Sénégal), Studia Informatica Universalis 10 (1), pp. 77-97.
- Taillandier, P., Amouroux, E., Vo, D.A. and Olteanu-Raimond A.M. (2012), Using Belief Theory to formalize the agent behavior: application to the simulation of avian flu propagation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 575-587.
- Le, V.M., Adam, C., Canal, R., Gaudou, B., Ho, T.V. and Taillandier, P. (2012), Simulation of the emotion dynamics in a group of agents in an evacuation situation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 604-619.
- Nguyen Vu, Q. A., Canal, R., Gaudou, B., Hassas, S., Armetta, F. (2012), TrustSets - Using trust to detect deceitful agents in a distributed information collecting system. In: Journal of Ambient Intelligence and Humanized Computing, Springer-Verlag, Vol. 3 N. 4, p. 251-263.

2011 * Taillandier, P. & Therond, O. (2011), Use of the Belief Theory to formalize Agent Decision Making Processes : Application to cropping Plan Decision Making. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 138-142.

- Taillandier, P. & Amblard, F. (2011), Cartography of Multi-Agent Model Parameter Space through a reactive Dicotomous Approach. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 38-42.
- Taillandier, P. & Stinckwich, S. (2011), Using the PROMETHEE Multi-Criteria Decision Making Method to Define New Exploration Strategies for Rescue Robots', IEEE International Symposium on Safety, Security, and Rescue Robotics, Kyoto, Japon, pp. 321 - 326.

2010 * Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), A cluster-based approach for disturbed, spatialized, distributed information gathering systems, in 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, pp. 588-603.

- Nguyen, N.D., Taillandier, P., Drogoul, A. and Augier, P. (2010), Inferring Equation-Based Models from Agent-Based Models: A Case Study in Competition Dynamics.In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp. 413-427.
- Amouroux, E., Gaudou, B. Desvaux, S. and Drogoul, A. (2010), O.D.D.: a Promising but Incomplete Formalism For Individual-Based Model Specification. in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), pp. 1-4.
- Nguyen, N.D., Phan, T.H.D., Nguyen, T.N.A., Drogoul, A. and Zucker, J-D. (2010), Disk Graph-Based Model for Competition Dynamic, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen, T.K., Marilleau, N., Ho T.V. and El Fallah Seghrouchni, A. (2010), A metamodel for specifying collaborative simulation, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), `TrustSets` - Using trust to detect deceitful agents in a distributed information collecting system, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), the best student paper award.
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S., Armetta, F. and Stinckwich, S. (2010), Using trust and cluster organisation to improve robot swarm mapping, Paper to appear in 'Workshop on Robots and Sensors integration in future rescue INformation system' (ROSIN 2010).

2009 * Taillandier, P. and Buard, E. (2009), Designing Agent Behaviour in Agent-Based Simulation through participatory method. In 'The 12th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Nagoya, Japan, pp. 571–578.

- Taillandier, P. and Chu, T.Q. (2009), Using Participatory Paradigm to Learn Human Behaviour. In 'International Conference on Knowledge and Systems Engineering', Ha noi, Viet Nam, pp. 55–60.
- Gaudou, B., Ho, T.V. and Marilleau, N. (2009), Introduce collaboration in methodologies of modeling and simulation of Complex Systems. In 'International Conference on Intelligent Networking and Collaborative Systems (INCOS '09)'. Barcelona, pp. 1–8.

- Nguyen, T.K., Gaudou B., Ho T.V. and Marilleau N. (2009), Application of PAMS Collaboration Platform to Simulation-Based Researches in Soil Science: The Case of the MIcro-ORganism Project. In 'IEEE International Conference on Computing and Telecommunication Technologies (IEEE-RIVF 09)'. Da Nang, Viet Nam, pp. 296–303.

- Nguyen, V.Q., Gaudou B., Canal R., Hassas S. and Armetta F. (2009), Stratégie de communication dans un système de collecte d'information à base d'agents perturbés. In 'Journées Francophones sur les Systèmes Multi-Agents (JFSMA'09)'.

2008 * Chu, T.Q., Boucher, A., Drogoul, A., Vo, D.A., Nguyen, H.P. and Zucker, J.D. (2008). Interactive Learning of Expert Criteria for Rescue Simulations. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 127–138.

- Amouroux, E., Desvaux, S. and Drogoul, A. (2008), Towards Virtual Epidemiology: An Agent-Based Approach to the Modeling of H5N1 Propagation and Persistence in North-Vietnam. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 26–33.