

Full Documentation of GAMA 1.6.1

Provided by the GAMA development team

<http://code.google.com/p/gama-platform/>

Table of Contents

Introduction.....	15
Introduction.....	15
Documentation.....	15
Source Code.....	16
Copyright Information.....	16
Developers.....	16
Citing GAMA.....	17
Contact Us.....	17
Platform.....	18
Platform.....	19
Platform.....	19
1.1 Installation.....	20
Installation.....	20
System Requirements.....	20
Installation of Java.....	20
1.2 Launching GAMA.....	22
Launching GAMA.....	22
Launching the Application.....	22
Choosing a Workspace.....	23
Welcome Page.....	24
1.3 Headless Mode.....	26
Headless Mode.....	26
Command.....	26
Experiment Input File.....	27

Output Directory.....	28
Simulation Output.....	29
Snapshot files.....	30
1.4 Updating GAMA.....	31
Updating GAMA.....	31
Manual Update.....	31
Automatic Update.....	34
1.5 Installing Plugins.....	37
Installing Plugins.....	37
Installation.....	37
Selected Plugins.....	42
1.6 Troubleshooting.....	46
Troubleshooting.....	46
On Ubuntu (& Linux Systems).....	46
On Windows.....	47
On MacOS X.....	47
Memory problems.....	47
Submitting an Issue.....	48
2. Workspace, Projects and Models.....	52
Workspace, Projects and Models.....	52
2.1 Navigating in the Workspace.....	53
Navigating in the Workspace.....	53
The Different Categories of Models.....	53
Moving Models Around.....	63
Closing and Deleting Projects.....	65
2.2 Changing Workspace.....	69
Changing Workspace.....	69
Switching to another Workspace.....	69
Cloning the Current Workspace.....	71
2.3 Importing Models.....	74
Importing Models.....	74
The "Import..." Menu Command.....	74
SVN import.....	78
Silent import.....	80

Drag'n Drop / Copy-Paste Limitations.....	80
2.4 Sharing Models.....	81
Sharing Models.....	81
Project Sharing.....	81
Committing Local Changes.....	88
Sharing project within GAMA.....	94
3. Editing Models.....	95
Introduction.....	95
3.1 GAML Editor.....	96
The GAML Editor.....	96
Creating a first model.....	96
Status of models in editors.....	99
Editor Preferences.....	104
Structural highlighting.....	106
Multiple editors.....	108
Local history.....	109
3.2 Validation of Models.....	113
Validation of Models.....	113
Syntactic errors.....	113
Semantic errors.....	114
Semantic warnings.....	116
Semantic information.....	117
Semantic documentation.....	118
Changing the visual indicators.....	118
Errors in imported files.....	120
Cleaning models.....	121
3.3 Graphical Editor.....	124
The Graphical Editor.....	124
Installing the graphical editor.....	124
Creating a first model.....	125
Status of models in editors.....	127
Diagram definition framework.....	128
Features.....	129
Pictogram color modification.....	141

GAML Model generation.....	141
4. Running Experiments.....	142
Running Experiments.....	142
4.1 Launching Experiments.....	143
Launching Experiments from the User Interface.....	143
From an Editor.....	143
From the Navigator.....	145
Running Experiments Automatically.....	146
4.2 Experiments User interface.....	148
Experiments User Interface.....	148
4.2.1 Menus and commands.....	150
Menus and Commands.....	150
Experiment Menu.....	150
Agents Menu.....	151
General Toolbar.....	152
4.2.2 Parameters view.....	155
Parameters View.....	155
Built-in parameters.....	155
Parameters View.....	155
Modification of parameters values.....	157
4.2.3 Inspectors and monitors.....	158
Inspectors and monitors.....	158
Agent Browser.....	158
Agent Inspector.....	159
Monitor.....	160
4.2.4 Displays.....	162
Displays.....	162
Classical displays (java2D).....	162
OpenGL displays.....	163
4.2.5 Batch Specific UI.....	165
Batch Specific UI.....	165
Information bar.....	165
Batch UI.....	165
4.2.6 Errors View.....	167

Errors View.....	167
5. Preferences.....	169
Preferences.....	169
Opening Preferences.....	169
General.....	169
Display.....	171
Editor.....	172
External.....	173
Advanced Preferences.....	175
GAML (GAMA Modeling Language).....	176
GAML (GAMA Modeling Language).....	177
GAML.....	177
1. Key Concepts.....	178
Key Concepts (Under construction).....	178
Lexical semantics of GAML.....	178
Translation into a concrete syntax.....	181
Vocabulary correspondance with the object-oriented paradigm as in Java.....	181
Vocabulary correspondance with the agent-based paradigm as in ! NetLogo.....	182
1.1 Runtime Concepts.....	184
Runtime Concepts (Under Construction).....	184
Agents Creation.....	184
Agents Step.....	184
Scheduling of Agents.....	185
2. Organization of a Model.....	187
Organization of a model (Under construction).....	187
Model Header (_model species_).....	187
Species declarations.....	188
Experiment declarations.....	188
3. Defining Species.....	189
Defining Species.....	189
Species Declaration.....	189
Scheduling Description.....	190

Topology Description.....	190
Types of Species.....	191
3.1 Global Species.....	192
Global Species (Under Construction).....	192
Declaration.....	192
Environment Size.....	193
Built-in Attributes.....	194
Built-in Actions.....	195
Scheduling.....	196
3.3 Grid Species.....	197
Grid Species.....	197
Declaration.....	197
Built-in variables.....	199
Access to cells.....	200
3.4 Graph Species.....	201
Graph Species (Under Construction).....	201
Declaration.....	201
Example.....	202
3.5 Mirror Species.....	203
Mirror Species (Under Construction).....	203
Declaration.....	203
Example.....	204
3.6 Multi-level Architecture.....	205
Multi-level Architecture (Under Construction).....	205
Details.....	205
Declaration of micro-species.....	205
Representation of an entity as different types of agent.....	206
Dynamic migration of agents.....	206
Access to micro-agents, host agent.....	207
3.7 Attaching Skills and Control.....	208
Attaching Skills and Control.....	208
Skills.....	208
Control Architecture.....	209
3.8 Defining Attributes.....	210

Defining Attributes (Under Construction).....	210
Attribute Declaration.....	210
Naming variables.....	213
3.9 Defining Actions.....	214
Defining Actions (Under Construction).....	214
Declaring Actions.....	214
Calling Actions.....	215
3.10 Defining Behaviors.....	216
Defining Behaviors (Under Construction).....	216
reflex.....	216
Init.....	216
3.11 Defining Equations.....	218
Defining equations (Under Construction).....	218
Equations.....	218
Classical Equations.....	219
3.12 Defining Aspects.....	221
Defining Aspects (Under Construction).....	221
Draw Command.....	221
Shape Primitive.....	222
3.13 Defining User Commands.....	224
Defining User Commands.....	224
Syntax.....	224
user_location.....	225
user_input operator.....	225
4. Defining GUI Experiments.....	227
Defining Experiments (under construction).....	227
4.1 Defining Parameters.....	228
Defining Parameters (Under Construction).....	228
4.2 Defining Outputs.....	229
Defining Outputs (Under Construction).....	229
4.2.1 Defining Displays.....	230
Defining Displays.....	230
4.2.1.1 Defining Chart Layers.....	231
Chart Layer (Under construction).....	231

Chart type.....	231
Data definition.....	232
Series/xy/scatter charts details.....	233
4.2.1.2 Defining Other Layers.....	235
Defining Other Layers.....	235
agents layer.....	235
species layer.....	236
image layer.....	237
text layer.....	238
graphics layer.....	238
4.2.1.3 3D Specific Instructions.....	239
3D Specific Instructions (Under construction).....	239
OpenGL display.....	239
Camera.....	240
Dynamic camera.....	240
Lighting.....	241
4.2.2 Defining Inspectors and Monitors.....	243
Defining Inspectors and Monitors (Under Construction).....	243
Monitors.....	243
Inspectors.....	243
4.2.3 Defining Files.....	244
Defining Files.....	244
5. Defining Batch Experiments.....	245
Defining Batch Experiments.....	245
The batch experiment facets.....	245
Action _step.....	246
Reflexes.....	246
Permanent.....	246
5.1 Parameter Space.....	248
Parameter Space.....	248
Parameter definition.....	248
The method element.....	248
5.2 Exploration Methods.....	250
Exploration Methods.....	250

Exhaustive exploration of the parameter space.....	250
Hill Climbing.....	251
Simulated Annealing.....	251
Tabu Search.....	252
Reactive Tabu Search.....	253
Genetic Algorithm.....	254
6. GAML Reference.....	256
Gaml Reference.....	256
6.1 Built-in Species.....	257
Built-in Species.....	257
Table of Contents.....	257
agent.....	257
AgentDB.....	258
base_edge.....	259
experiment.....	259
graph_edge.....	260
graph_node.....	260
model.....	260
multicriteria_analyzer.....	261
Physical3DWorld.....	261
6.1.1 'agent'.....	262
The 'agent' built-in species (Under Construction).....	262
`agent` attributes.....	262
`agent` actions.....	262
6.1.2 'model'.....	263
The 'model' built-in species (Under Construction).....	263
`model` attributes.....	263
`model` actions.....	263
6.1.3 'experiment'.....	264
The 'experiment' built-in species (Under Construction).....	264
`experiment` attributes.....	264
`experiment` actions.....	264
6.2 Built-in Skills.....	265
Built-in Skills.....	265

Table of Contents.....	266
advanced_driving.....	266
busTransportation.....	268
communicating.....	269
driving.....	272
driving2d.....	274
graphic.....	275
grid.....	276
humanmoving.....	276
MAELIA.....	278
MDXSKILL.....	279
moving.....	279
moving3D.....	281
network.....	282
optimizing.....	283
physical3D.....	283
roadTrafficManagement.....	284
skill_road.....	284
skill_road_node.....	285
socket.....	285
SQLSKILL.....	287
trafficMoving.....	288
trafficMoving.....	289
6.3 Built-in Control Architectures.....	291
Built-in Control Architectures.....	291
Attachment of Control Architecture.....	291
List of Control Architectures.....	291
6.3.1 Finite State Machine.....	293
Finite State Machine.....	293
Declaration.....	293
State.....	293
6.3.2 Task Based.....	296
Task Based.....	296
Declaration.....	296

Task.....	296
6.3.3 User Controlled.....	298
User Controlled.....	298
Declaration.....	298
User Panel.....	298
User Controlled.....	300
6.4 Statements.....	301
Statements.....	301
Table of Contents.....	301
Statements by kinds.....	301
Statements by embedment.....	302
General syntax.....	303
6.5 Data Types.....	386
Types (Under Construction).....	386
Primitive built-in types.....	387
Complex built-in types.....	388
Defining custom types.....	396
6.5.1 File Types.....	399
File Types.....	399
Text File.....	400
CSV File.....	401
Shapefile.....	402
OSM File.....	404
Grid File.....	406
Image File.....	408
SVG File.....	409
Property File.....	410
R File.....	411
3DS File.....	412
OBJ File.....	413
6.6 Expressions.....	414
Expressions.....	414
6.6.1 Literals.....	415
Literals.....	415

Simple Types.....	415
Literal Constructors.....	415
Universal Literal.....	416
6.6.2 Units and constants.....	417
Units and constants.....	417
Table of Contents.....	418
Constants.....	418
Graphics units.....	418
Length units.....	419
Surface units.....	419
Time units.....	419
Volume units.....	419
Weight units.....	420
Colors.....	420
6.6.3 Pseudo-variables.....	424
Pseudo-variables.....	424
self.....	424
myself.....	424
each.....	425
6.6.4 Variables and attributes.....	426
Variables and Attributes.....	426
Direct Access.....	426
Remote Access.....	427
6.6.5 Operators (A to K).....	429
Operators (A to K).....	429
Definition	429
Priority between operators.....	430
Using actions as operators.....	430
Table of Contents.....	431
Operators by categories.....	431
Operators.....	438
6.6.6 Operators (L to Z).....	534
Operators (L to Z).....	534
Definition.....	534

Priority between operators.....	535
Using actions as operators.....	535
Table of Contents.....	536
Operators by categories.....	536
Operators.....	543
7. Recipes.....	621
Recipes.....	621
7.1 Optimizing Models.....	622
Optimizing Models.....	622
machine_time.....	622
Scheduling.....	622
Grid.....	623
Operators.....	623
Displays.....	625
7.4 Using Database Access.....	627
Using Database Access.....	627
Description.....	628
Supported DBMS.....	628
SQLSKILL.....	628
MDXSKILL.....	633
AgentDB.....	636
Using database features to define environment or create species.....	640
7.5 Calling R.....	644
Introduction.....	644
Configuration in GAMA.....	644
Calling R from GAML.....	644
7.8 Defining User Interaction.....	648
Defining User Interaction.....	648
Event.....	648
User Command.....	649
User Control Architecture.....	651
References.....	653
References.....	654

References.....	654
Papers about GAMA.....	654
PhD theses.....	655
Research papers that use GAMA as modeling/simulation support.....	655

—

Introduction

Introduction

—

GAMA is a simulation platform, which aims at providing field experts, modellers, and computer scientists with a complete modelling and simulation development environment for building spatially explicit multi-agent simulations. It has been first developed by the Vietnamese-French research team MSI (located at IFI, Hanoi, and part of the IRD/UPMC International Research Unit UMMISCO) from 2007 to 2010, and is now developed by a consortium of academic and industrial partners led by UMMISCO, among which the University of Rouen, France, the University of Toulouse 1, France, the University of Orsay, France, the University of Can Tho, Vietnam, the National University of Hanoi, EDF R&D, France, and CEA LISC, France. Some of the features of GAMA are illustrated in the videos below (more can be found [in our Youtube channel](#)).

Beyond these features, GAMA also offers:

- A complete modeling language, GAML, for modeling agents and environments
- A large and extensible library of primitives (agent's movement, communication, mathematical functions, graphical features, ...)
- A cross-platform reproducibility of experiments and simulations
- A powerful declarative drawing and plotting subsystem
- A flexible user interface based on the Eclipse platform
- A complete set of batch tools, allowing for a systematic or "intelligent" exploration of models parameters spaces

—

Documentation

The documentation of GAMA is available online on the wiki of the project. It is organized around a few central activities ([installing GAMA](#) , [G__WritingModels writing models], [running experiments](#) , [G__DevelopingExtensions developing new extensions to the platform]) and provides complete references on both the [GAML language](#) , the [platform](#) itself, and the scientific aspects of our work (with a complete [bibliography](#)). Several [G__Tutorials tutorials] are also provided in the documentation in order to minimize the learning curve, allowing users to build, step by step, the models corresponding to these tutorials, which are of course shipped with the platform. The documentation can be accessed from the side bar of this page. A good starting point for new users is [the installation page](#) .

Source Code

GAMA can be [G__Downloads downloaded] as a regular application or [built from source](#), which is necessary if you want to contribute to the platform. The source code is available from this read-only SVN repository: svn checkout <http://gama-platform.googlecode.com/svn/branches/current>. Which you can also browse from the web [here](#). It is, in any case, recommended to follow the instructions on [G__InstallingSvnVersion this page] in order to build GAMA from source.

Copyright Information

This is a free software (distributed under the GNU GPL v3 license), so you can have access to the code, edit it and redistribute it under the same terms. Independently of the licensing issues, if you plan on reusing part of our code, we would be glad to know it !

Developers

GAMA is being designed, developed and maintained by an active group of researchers coming from different institutions in France and Vietnam. Please find below a short introduction to each of them and a summary of their contributions to the platform:

- **Alexis Drogoul**, Senior Researcher at the [IRD](#), member of the [UMMISCO](#) International Research Unit. Mostly working on agent-based modeling and simulation. Has contributed and still contributes to the original design of the platform, including the GAML language (from the meta-model to the editor) and simulation facilities like Java2D displays.
- **Patrick TAILLANDIER**, Associate Professor at the [University of Rouen](#), member of the [IDEES](#) CNRS Mixed Research Unit. Contributes since 2008 to the spatial and graph features (GIS integration, spatial operators) and to parameter space search algorithms. Currently working on new features related to graphical modeling and traffic simulation.
- **Benoit Gaudou**, Associate Professor at the [University Toulouse 1 Capitole](#), member of the [IRIT](#) CNRS Mixed Research Unit. Contributes since 2010 to documentation and unit test generation and coupling mathematical (ODE and PDE) and agent paradigms.
- **Arnaud Grignard**, software engineer and PhD fellow ([PDI-MSC](#)) at [UPMC](#). Contributes since 2011 to the development of new features related to visualization (3D Display), online analysis and interaction.
- **Huynh Quang Nghi**, software engineering lecturer at [CTU](#) and PhD fellow ([PDI-MSC](#)) at [UPMC](#). Contributes since 2012 to the development of new features related to GAML parser, coupling formalisms in EBM-ABM and ABM-ABM.
- **Truong Minh Thai**, software engineering lecturer at [CTU](#) and PhD fellow (PRJ322-MOET) at [IRIT - UT1](#). Contributes since 2012 to the development of new features related to data management and analysis.
- **Nicolas Marilleau**, Researcher at the [IRD](#), member of the [UMMISCO](#) International Research Unit and associate researcher at [DISC](#) team of [FEMTO-ST](#) institute. Contributes since 2010 to the development of headless mode and the high performance computing module.

- **Philippe Caillou** , Associate professor at the [University Paris Sud 11](#) , member of the [LRI](#) and [INRIA](#) project-team [TAO](#) . Contributes since 2012 and actually working on charts, simulation analysis and BDI agents.
 - **Vo Duc An** , Post-doctoral Researcher, working on synthetic population generation in agent-based modelling, at the [UMMISCO](#) International Research Unit of the [IRD](#) . Has contributed to bringing the platform to the Eclipse RCP environment and to the development of several features (e.g., the FIPA-compliant agent communication capability, the multi-level architecture).
 - **Truong Xuan Viet** , software engineering lecturer at [CTU](#) and PhD fellow ([PDI-MSC](#)) at [UPMC](#) . Contributes since 2011 to the development of new features related to R caller, online GIS (OPENGIS: Web Map Service - WMS, Web Feature Services - WFS, Google map, etc).
 - Samuel Thiriot
-

Citing GAMA

If you use GAMA in your research and want to cite it (in a paper, presentation, whatever), please use this reference:

Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, Alexis Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'The 16th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Dunedin, New Zealand, Volume 8291, pp. 242-258. or you can choose to cite the website instead:

GAMA Platform website, <https://code.google.com/p/gama-platform/> A complete list of references (papers and PhD theses on or using GAMA) is available on the [references](#) page.

Contact Us

The best way to get in touch with the developers of GAMA is to sign in for the [gama-platform@googlegroups.com mailing list](mailto:gama-platform@googlegroups.com) . If you wish to contribute to the platform, you might want, instead or in addition, to sign in for the [gama-dev@googlegroups.com mailing list](mailto:gama-dev@googlegroups.com) . On both lists, we generally answer quite quickly to requests. Finally, if you think you have found a bug in GAMA, or if you absolutely need a feature that does not exist yet, it is much more efficient and time-saving for everyone (including current and future users) to create a new issue report. Please refer to [these instructions](#) to do so.

Platform

— Platform

Platform

—

GAMA consists of a single application that is based on the RCP architecture provided by [Eclipse](#). Within this single application software, often referred to as a [_ platform](#), users can undertake, without the need of additional third-parties softwares, most of the activities related to modeling and simulation, namely [editing models](#) and [simulating, visualizing and exploring them](#) using dedicated tools. This allows for a very flexible workflow that can be adapted easily to the needs and habits of modelers, or even teams of modelers thanks to the integration of [model sharing tools](#). First-time users may however be intimidated by the apparent complexity of the platform, so this part of the documentation has been designed to ease their first contact with it, by clearly identifying tasks of interest to modelers and how they can be accomplished within GAMA. It is accomplished by firstly providing some background about important notions found throughout the platform, especially those of [workspace and projects](#) and explaining how to [organize and navigate through models](#). Then we take a look at the [edition of models](#) and its various tools and components ([dedicated editors](#), of course, but also [validators](#)). Finally, we show how to [run experiments](#) on these models and what support the [user interface](#) can provide to users in this task.

1.1 Installation

Installation

—
GAMA comes in 6 different versions (32 & 64 bits for Windows, Linux and MacOS). You first need to determine which version you are going 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 [G__Downloads Downloads page], expand the zip file wherever you want on your machine, and [launch GAMA](#) .

System Requirements

GAMA requires that Java 1.6 or 1.7 be installed on your machine, approximately 200MB of disk space is available and that a minimum of RAM (4GB is recommended) is installed.

Installation of Java

On Windows and Linux the recommended Java Virtual Machine under which GAMA has been tested is the one distributed by Oracle. On MacOS X, it is (still) the one distributed by Apple (see below). 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 it is not mandatory (GAMA should run fine, but slower, under a JRE).

On MacOS X (esp. Lion, Mountain Lion, Mavericks and Yosemite)

The latest version of GAMA requires a JVM (or JDK) compatible with Java 1.6 to run. There are two of them available for MacOS X. JDK 1.7 is distributed by Oracle, while JDK 1.6 is distributed by Apple. Because of this bug in SWT (https://bugs.eclipse.org/bugs/show_bug.cgi?id=374199), GAMA will not run correctly under JDK 1.7 (all the displays will appear empty). If JDK 1.7 is already installed, it is then necessary to also install the JDK 1.6 distributed by Apple in order to run GAMA. The latest version (as of October, 2013) can be obtained 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. If you upgrade your Mac OS X version to **Yosemite** (latest version, aka Mac OS X 10.10), some changes in the management of Java Virtual Machines might prevent the GAMA displays from working

and it will make your models crash the platform. One fix for this behavior is, after having upgraded Mac OS X, to install (or reinstall in case you have already installed it) « Java for OS X 2014-001 », available here: http://support.apple.com/kb/DL1572?viewlocale=en_US&locale=en_US .

On Windows 7 & 8 64 bits

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: [Java download section](#)
- Execute the downloaded file
- You can check that a **Java\jre7** 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**

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
```

You can then switch between java version using:

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

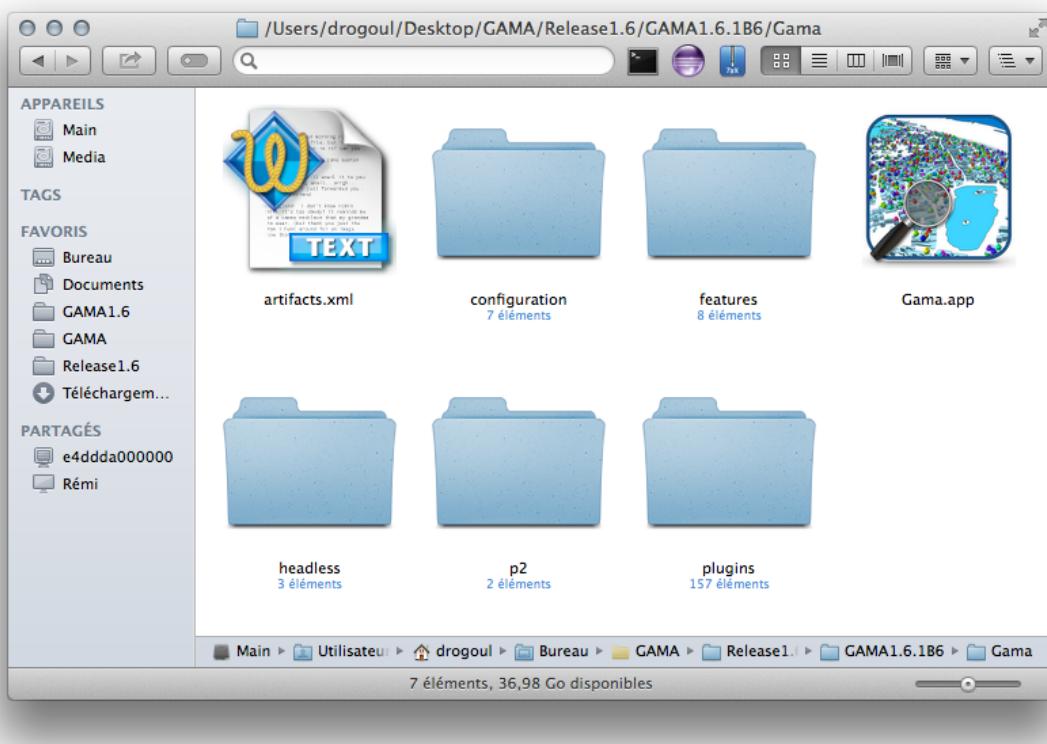
see [the troubleshooting page](#) for more information on workaround for problems on Unbuntu.

1.2 Launching GAMA

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, or if error messages appear, please refer to the [installation](#) or [troubleshooting](#) instructions.

Launching the Application



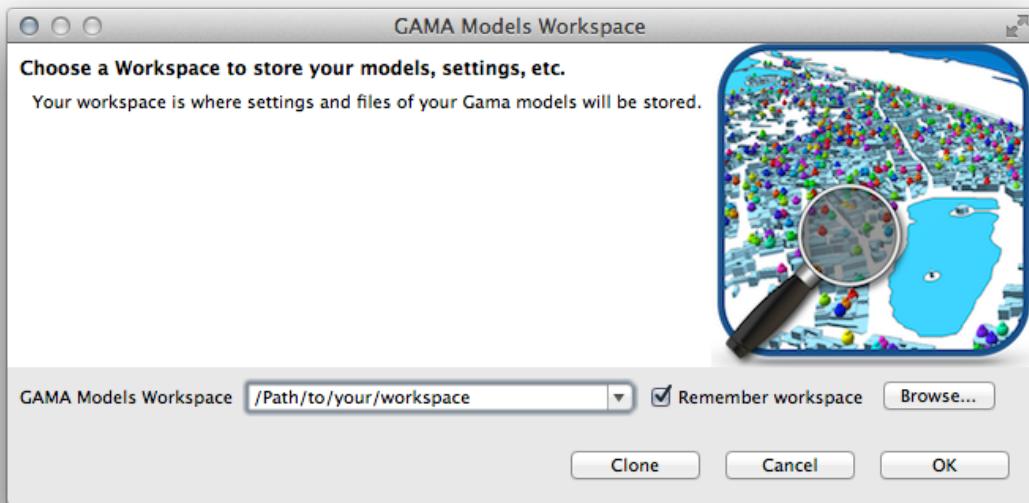
Note that GAMA can also be launched in two different other ways:

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](#).

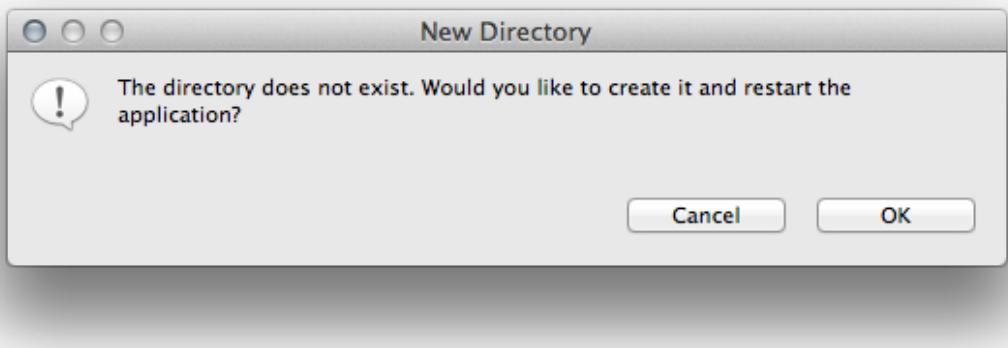
2. From the terminal, using a path to a model file and the name or number of an experiment, in order to allow running this experiment directly (note that the two arguments are optional: if the second is omitted, the file is imported in the workspace if not already present and opened in an editor; if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

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.

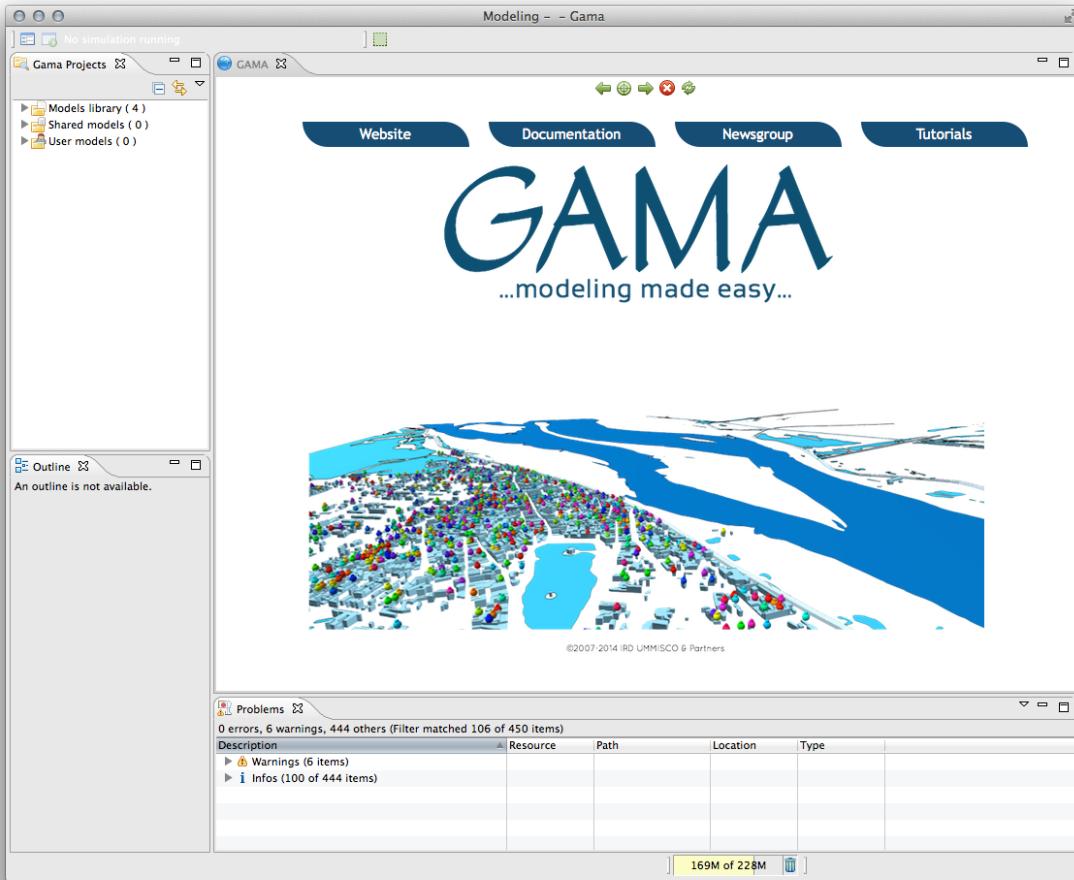


You can enter its address or browse your filesystem using the appropriate button. If the folder already exists, it will be reused (after a warning if it is not already a workspace). If not, it will be created. It is always a good idea, when you launch a new version of GAMA for the first time, to create 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.

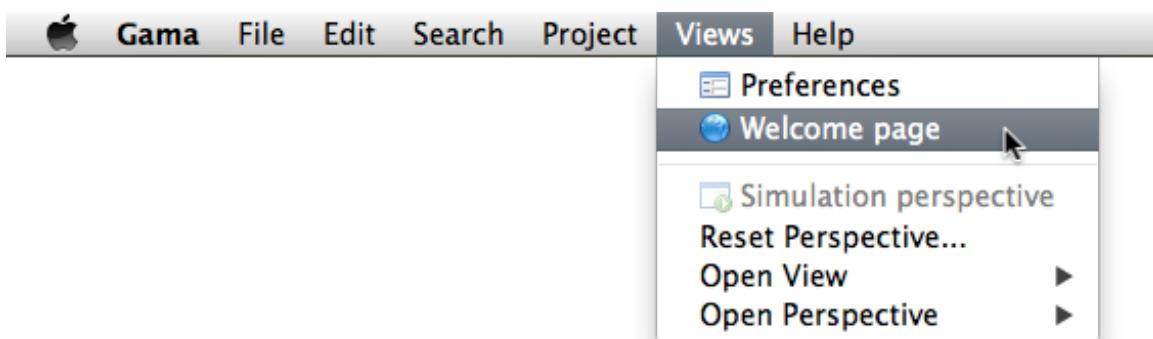


Welcome Page

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](#).

1.3 Headless Mode

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.

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 gamaHeadless.sh on MacOSX and Linux, and gamaHeadless.bat on Windows.

```
sh gamaHeadless.sh $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)
- 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). Note that the current directory to run gamaHeadless 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"?>
<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>
```

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

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.

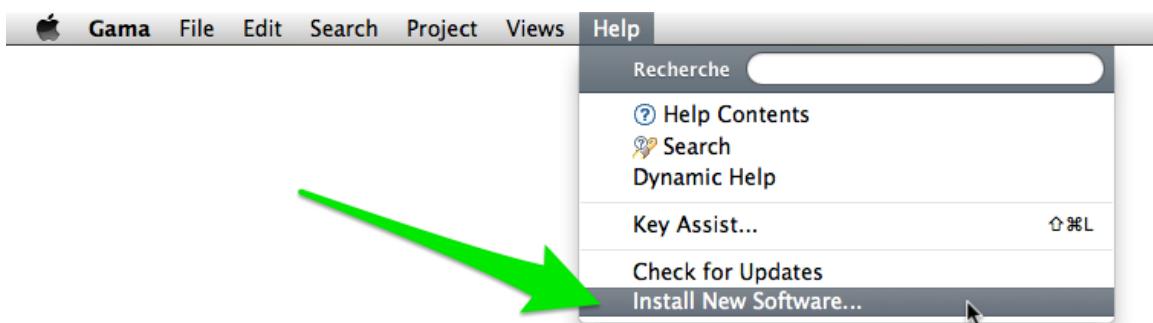
1.4 Updating GAMA

Updating GAMA

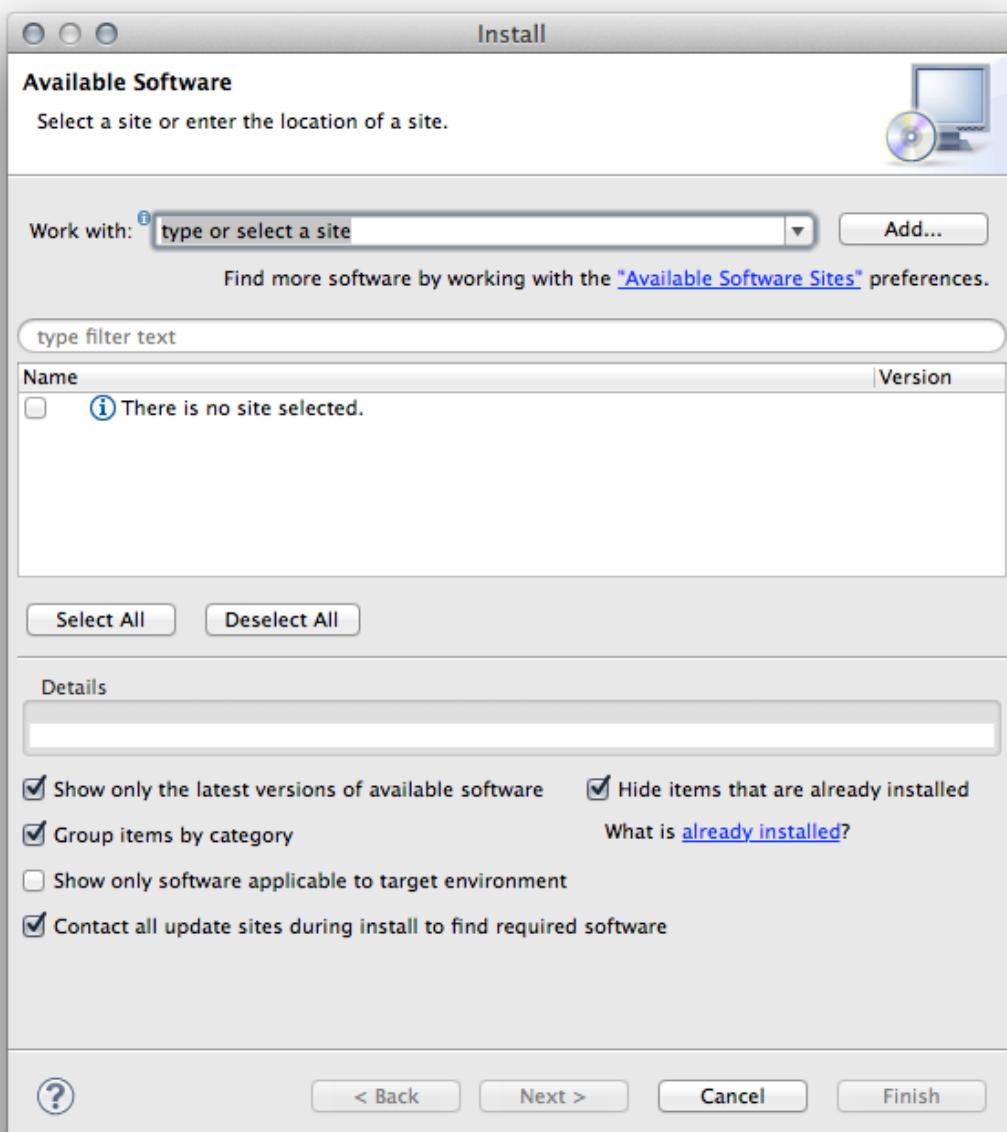
Unless you are using the version of GAMA built from the sources available in the SVN repository of the project (see [G__InstallingSvnVersion here]), you are normally running a specific **release** of GAMA that sports a given **version number** (e.g. GAMA 1.6.1, GAMA 1.7, etc.). When new features were 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.

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.



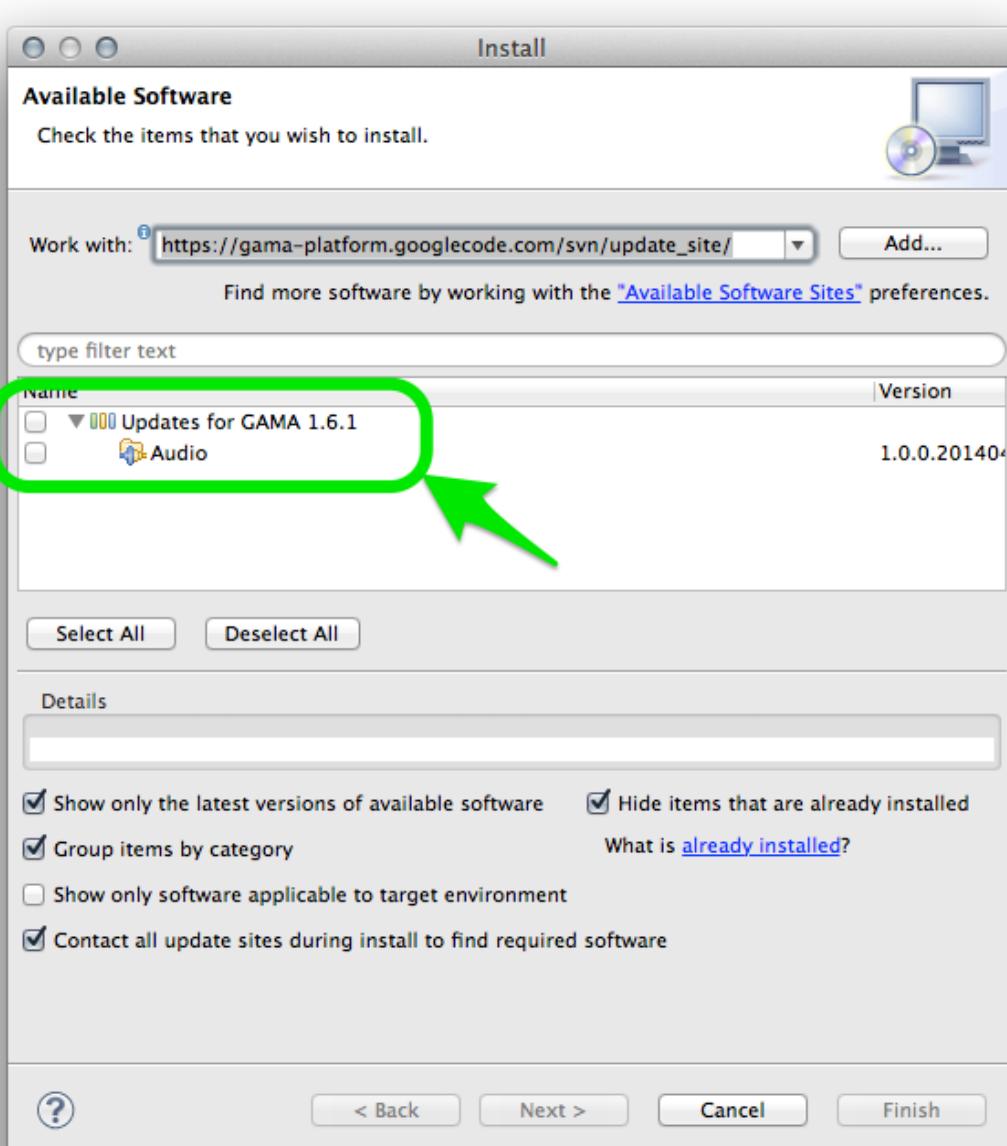
In general, it is preferable to use the second command, as more options (including that of *desinstalling* some plugins) are provided. Once invoked, it makes the following dialog appear:



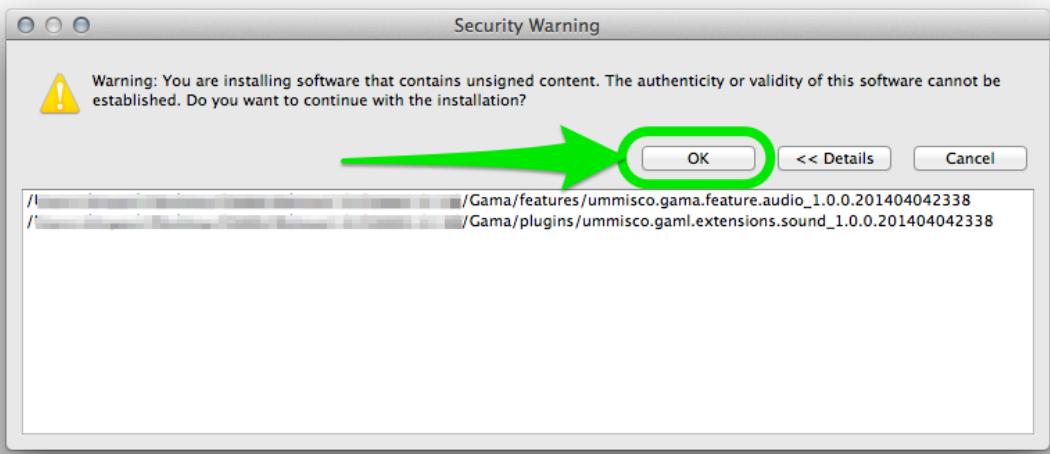
GAMA expects the user to enter a so-called *update site*. You can copy and paste the following line (or choose it from the drop-down menu as this address is built inside GAMA):

```
https://gama-platform.googlecode.com/svn/update_site/
```

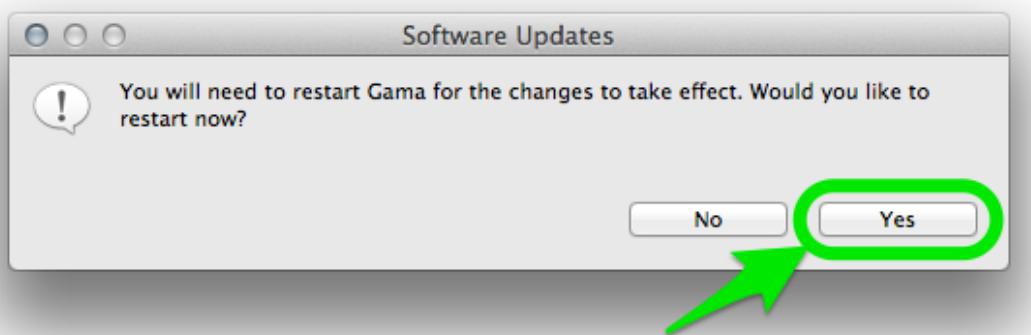
GAMA will then scan the entire update site, looking both for new plugins (the example below) and updates to existing plugins. The list available in your installation will of course be different from the one displayed here.



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 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".

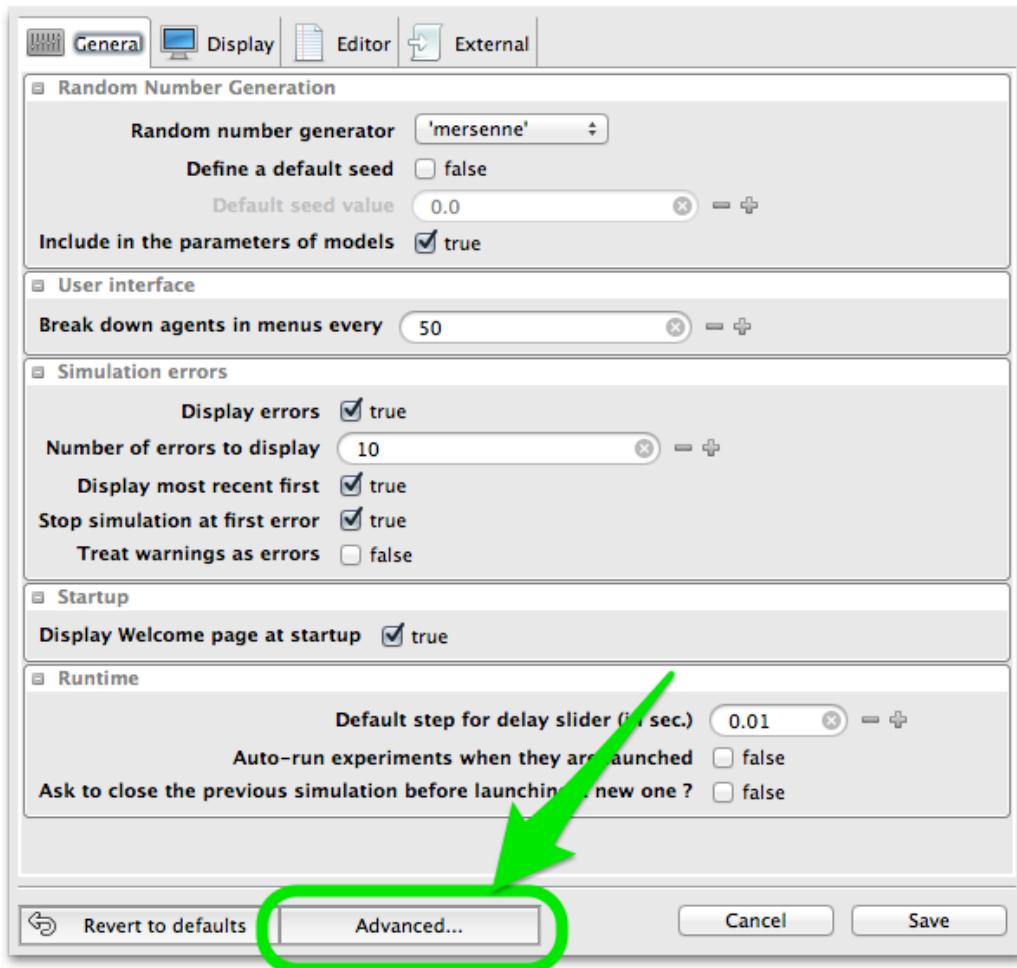


Once the plugins are installed, GAMA will ask you whether you want to restart or not. It is always safer to do so, so select "Yes" and let it close by itself, register the new plugins and restart.

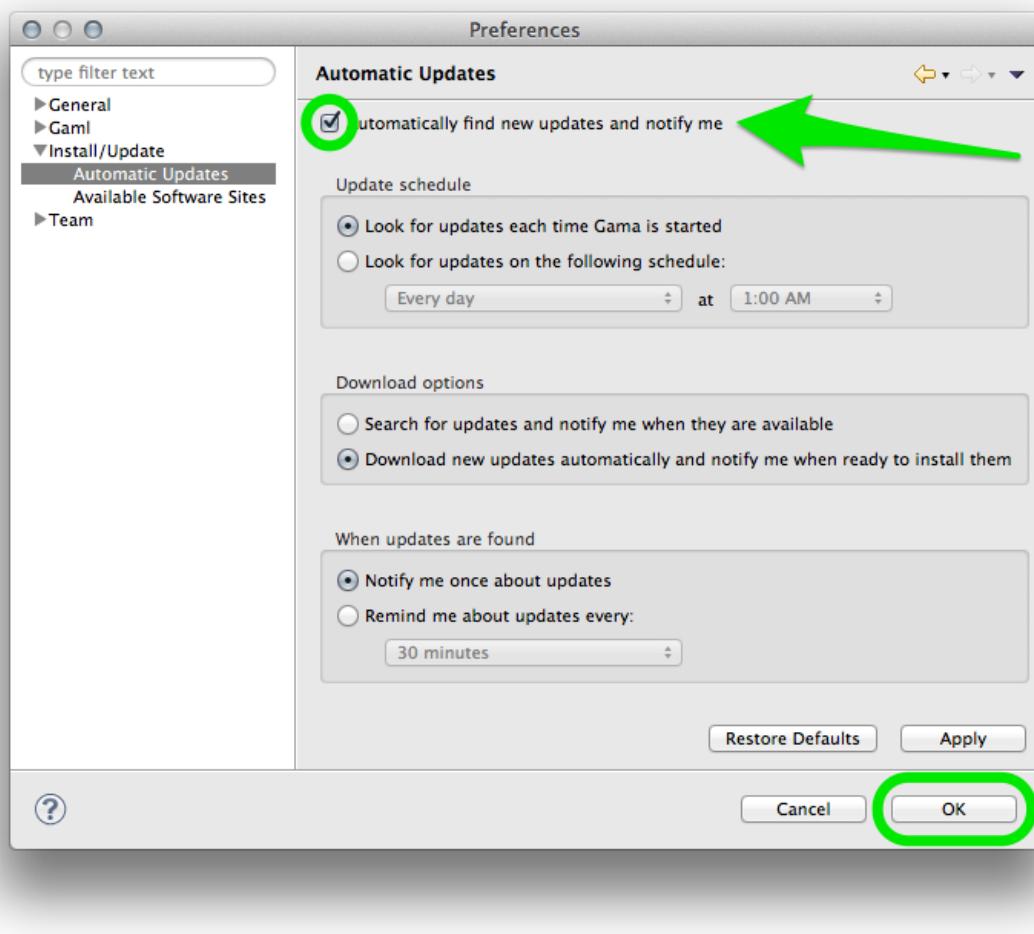


Automatic Update

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.

1.5 Installing Plugins

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:

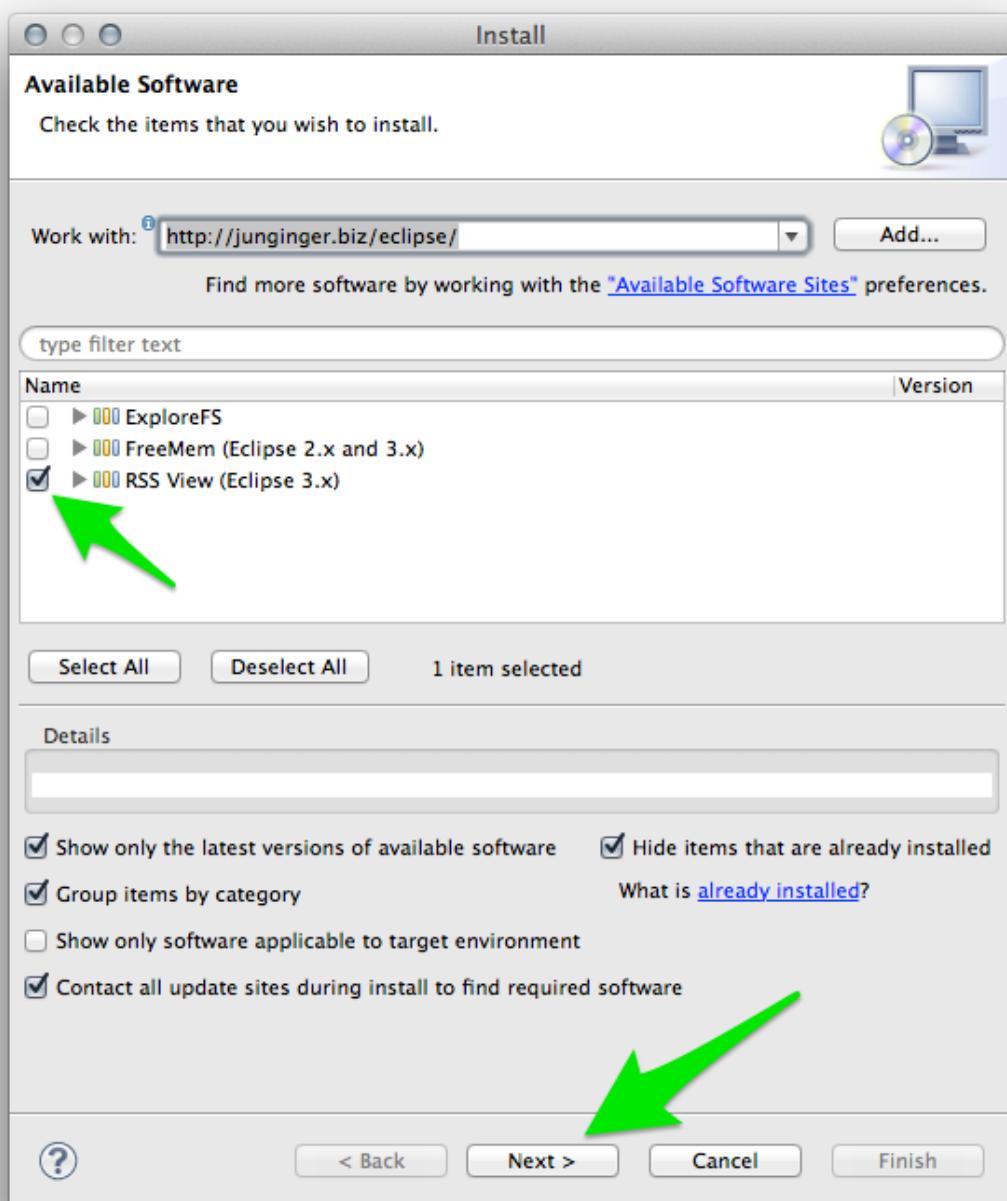
1. The current version of GAMA is based on Eclipse Juno (version number 3.8.2), which excludes de facto all the plugins targeting solely the 4.3 (Kepler) or 4.4 (Luna) versions of Eclipse. These will refuse to install anyway.
2. The Eclipse foundations in GAMA are only a subset of the complete Eclipse 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.

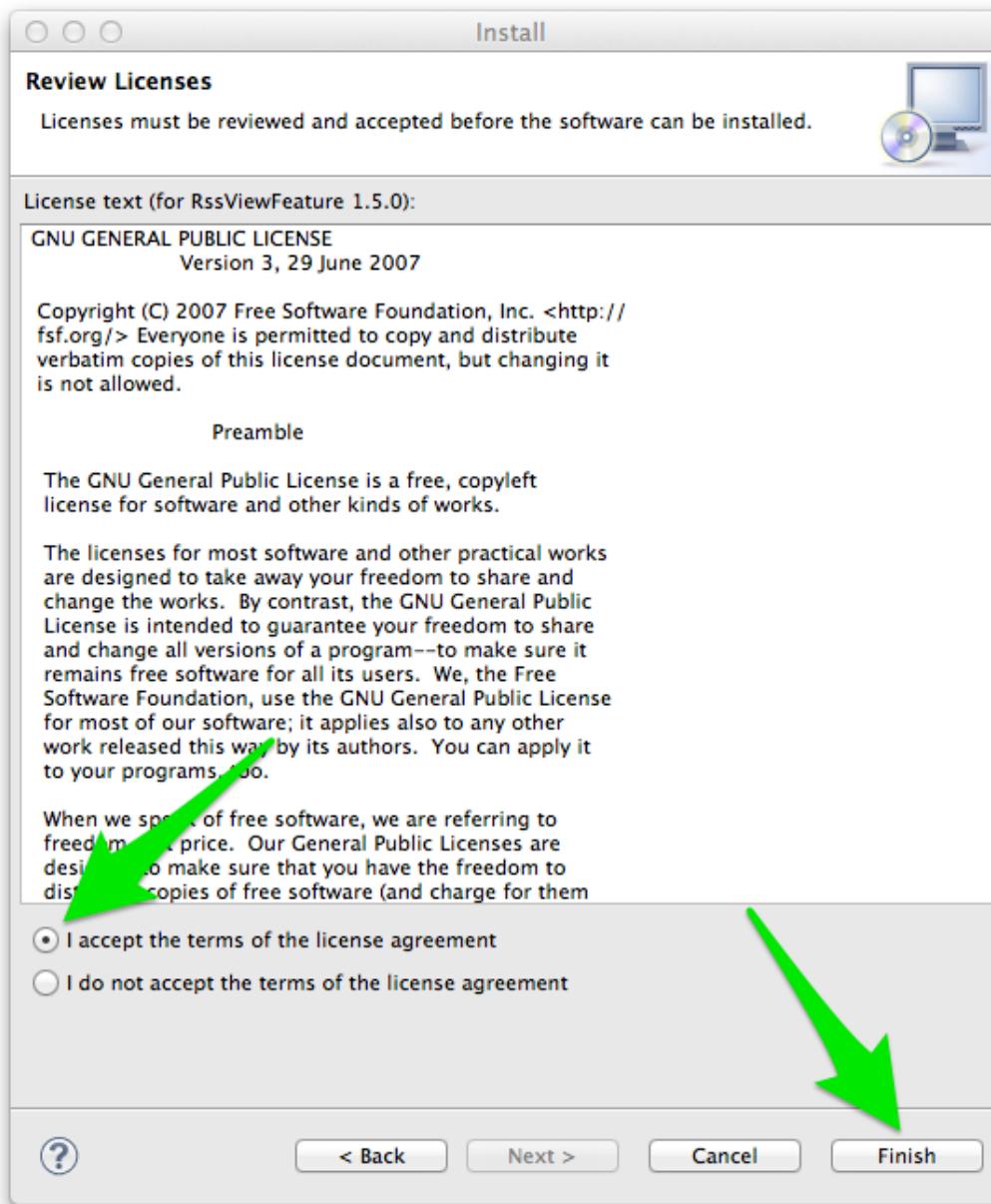
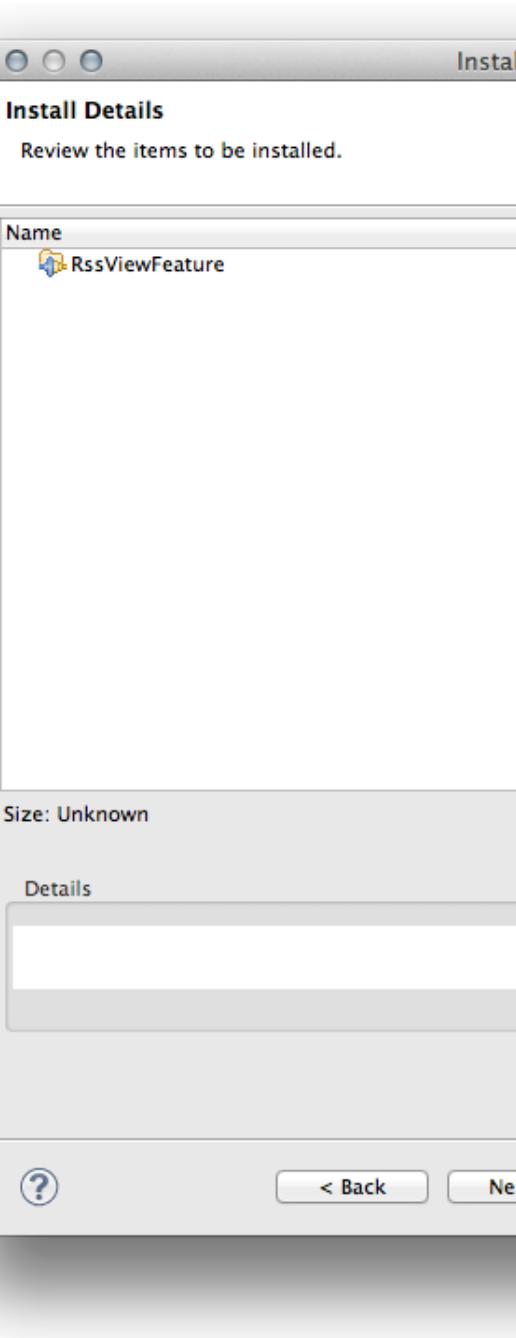
Installation

Installing new plugins is a process identical to the one described when [G__UpdatingGama 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 :

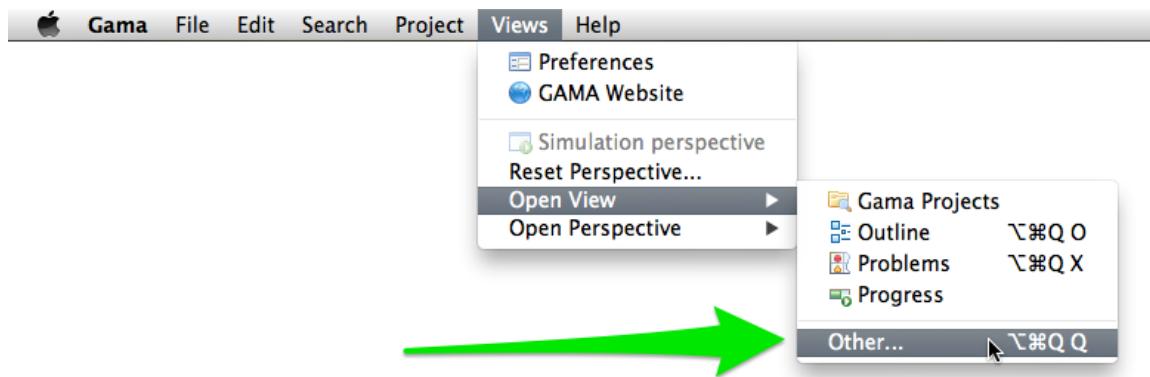
All plugins are installed with the standard update manager of Eclipse. It will guide you through the installation process and also eases keeping your plugins up-to-date. Just add the update site: <http://www.junginger.biz/eclipse/> So we just have to follow these instructions, which leads us to the following dialog, in which we select "RSS view" and click "Next".



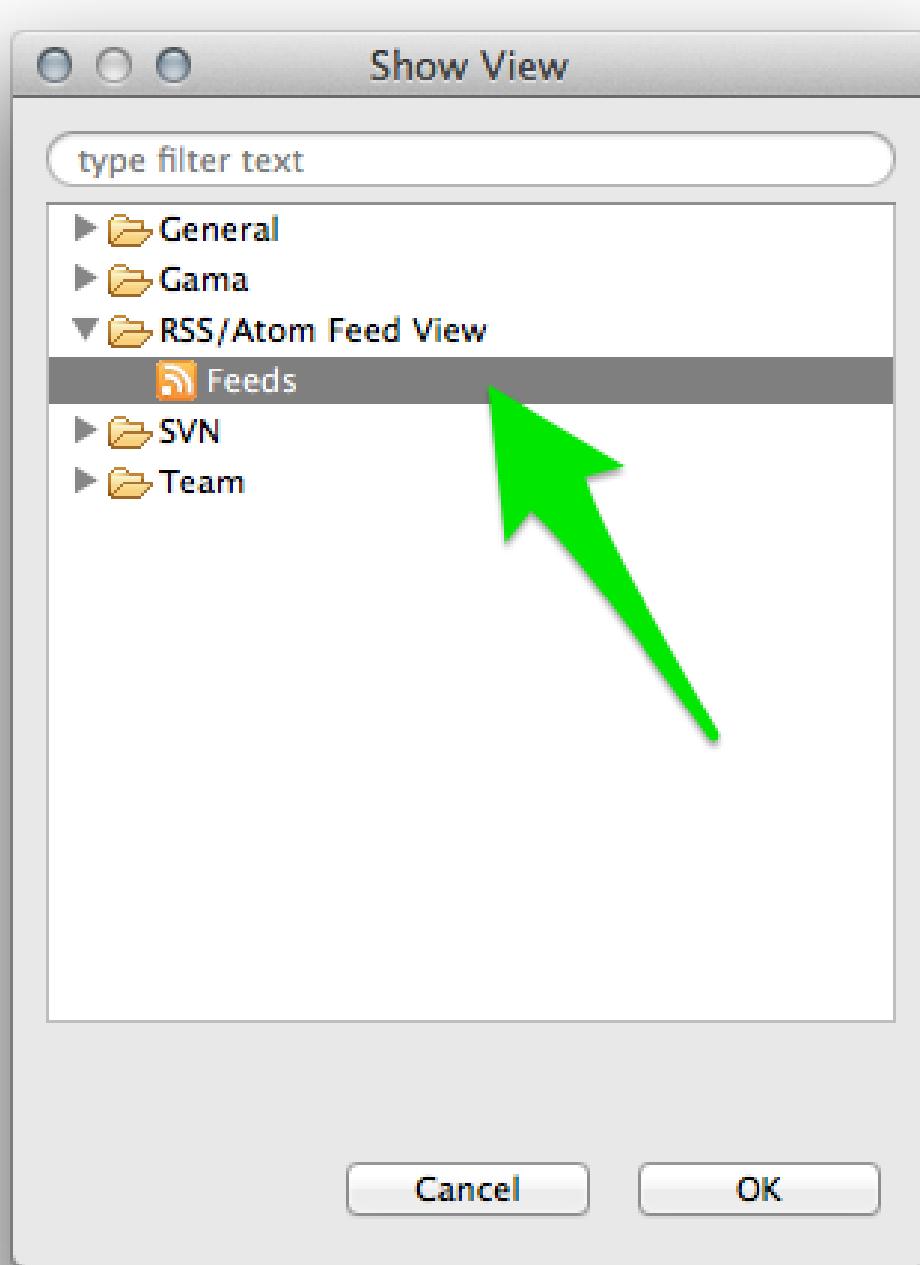
The initial dialog is followed by two other ones, a first to report that the plugin satisfies all the dependencies, a second to ask the user to accept the license agreement.



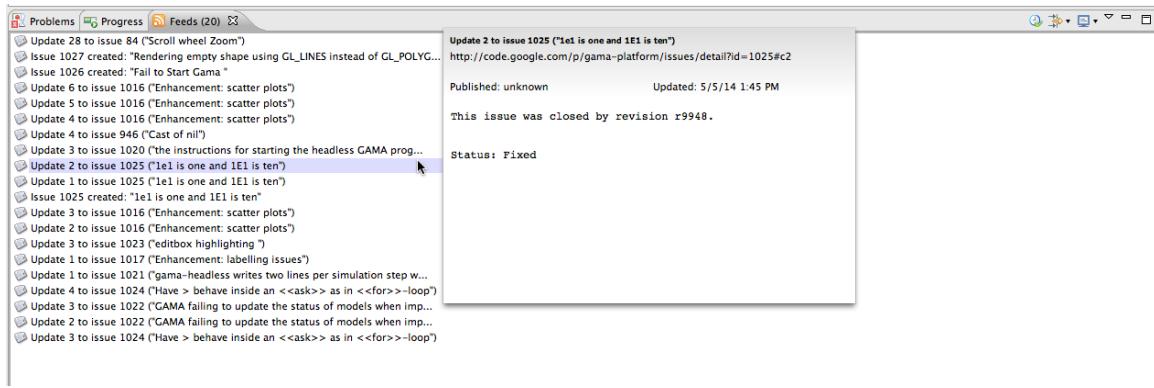
Once we dismiss the warning that the plugin is not signed and accept to restart GAMA, we can test the new plugin by going to the "Views" menu.



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

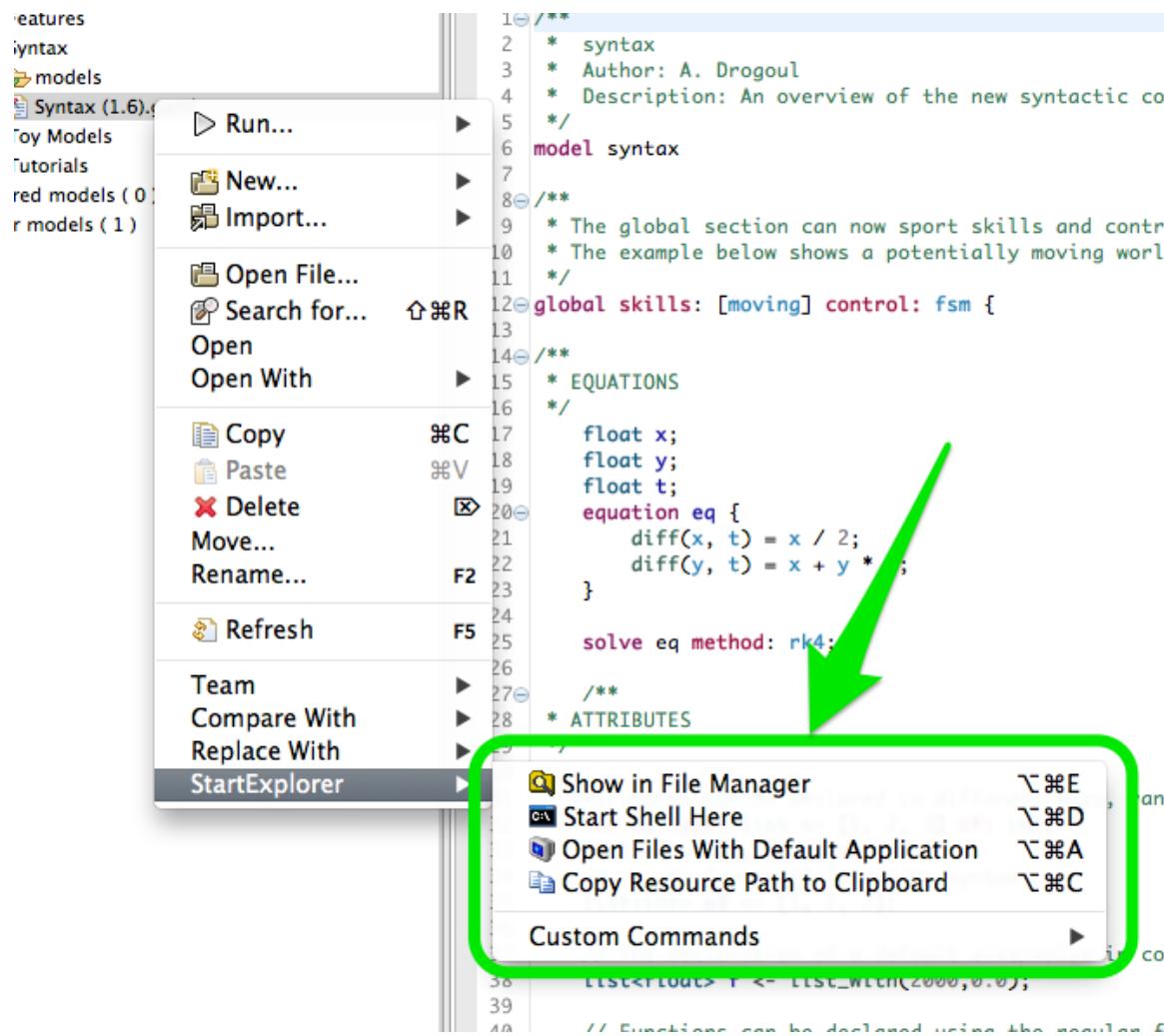
In addition to the RSS reader described above, below is a list of plugins that have been tested to work with GAMA. There are many others so take the time to explore them !

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

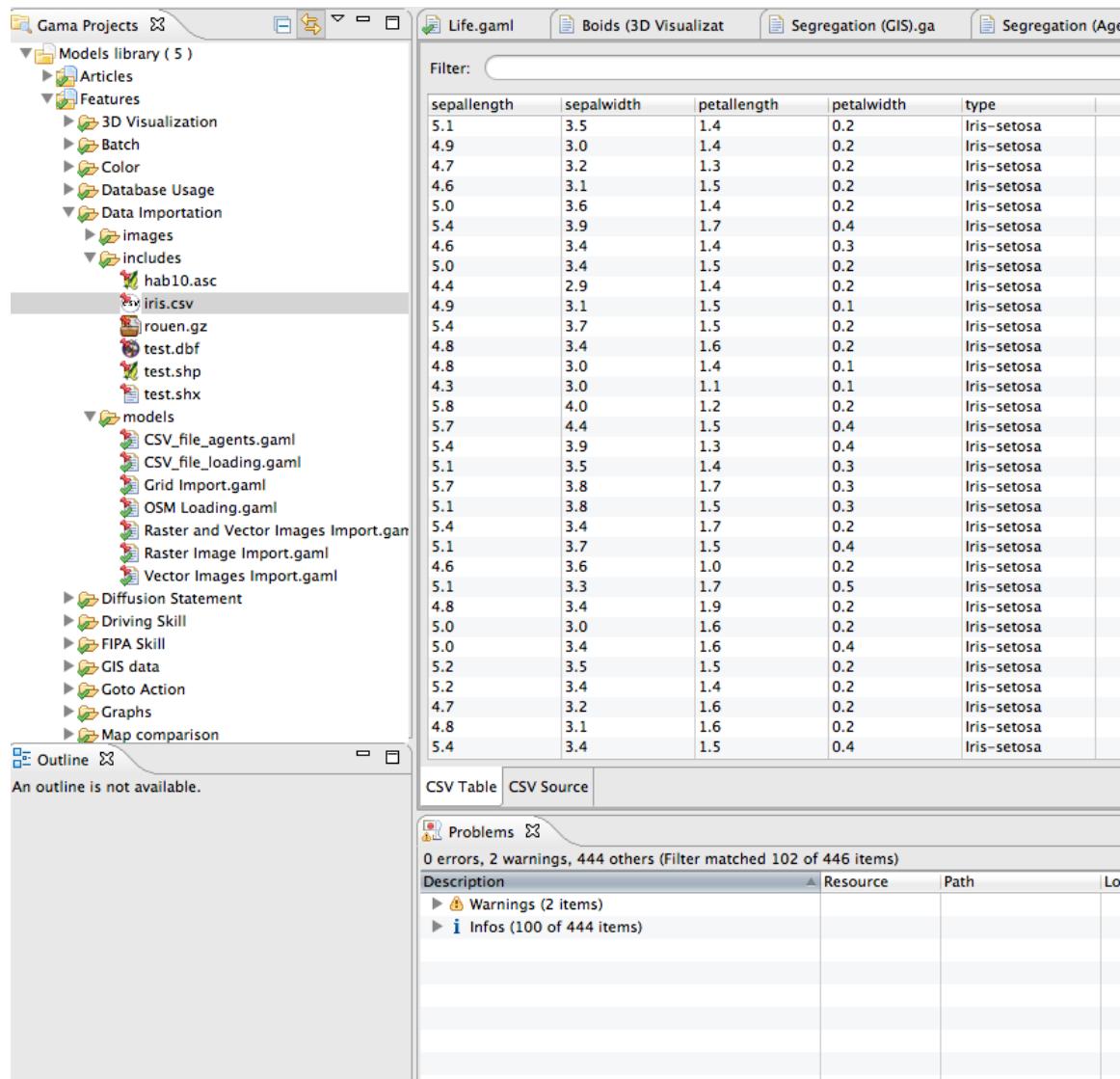
Startexplorer

- 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/>



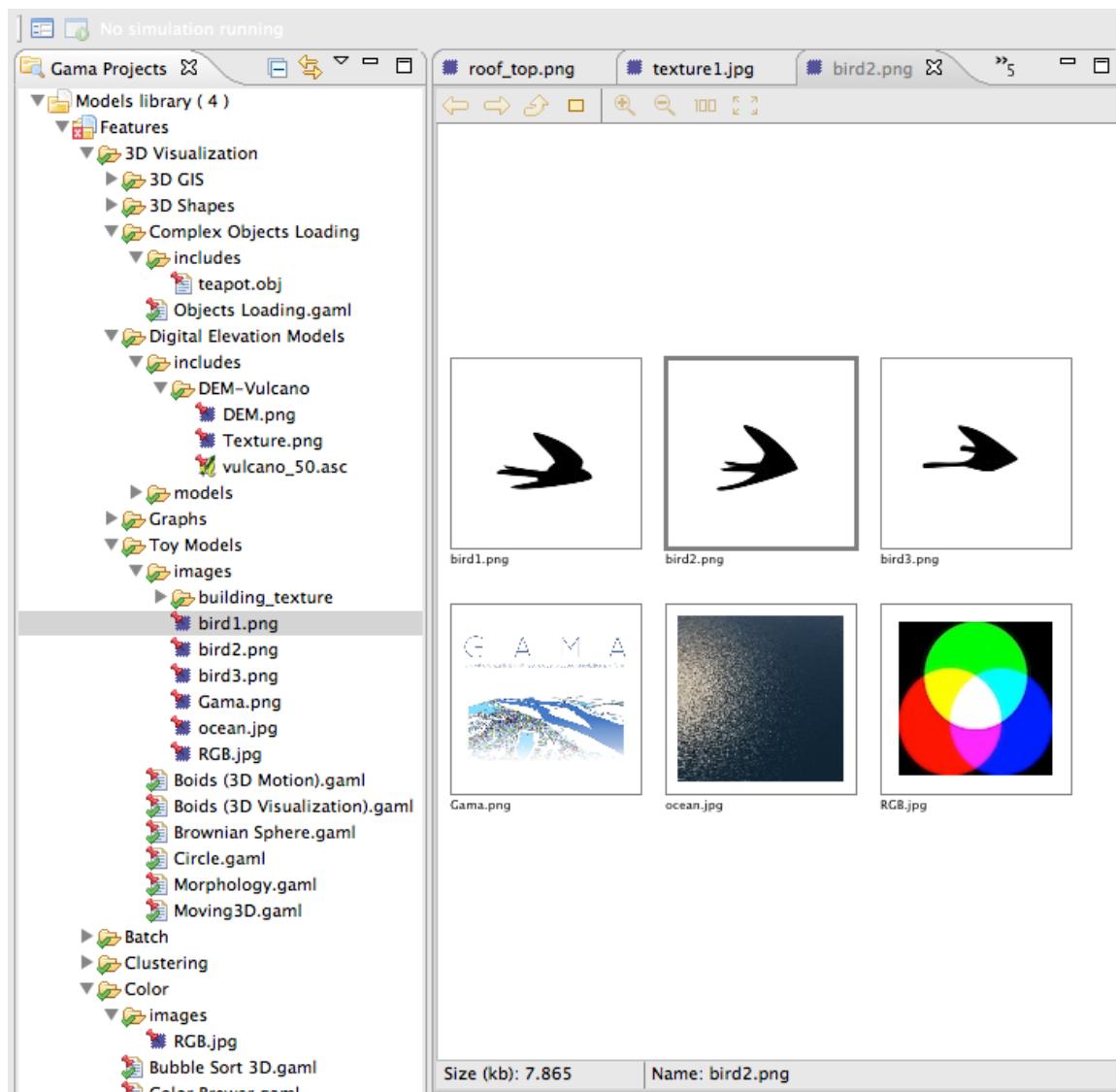
CSV Edit

- 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>



1.6 Troubleshooting

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.

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.

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_MENU_PROXY=0
./Gama
```

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

On Windows

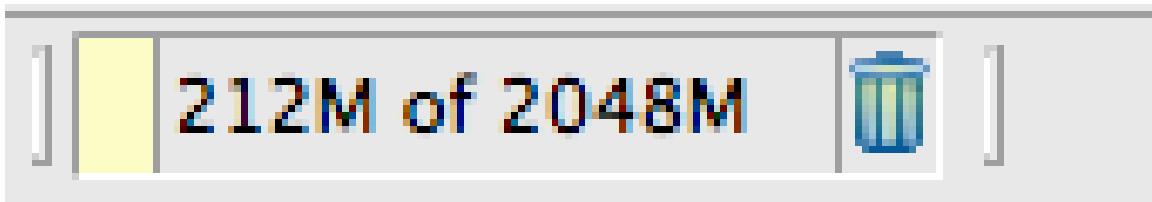
No common trouble...

On MacOS X

No common trouble...

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.



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:

```
1 -startup
2 ../../plugins/org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar
3 --launcher.library
4 ../../plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.200.v20120913-144807
5 -nl
6 ${target.nl}
7 -data
8 @noDefault
9 --launcher.defaultAction
10 openFile
11 -vmargs
12 -server
13 -Xverify:none
14 -Xms256m
15 -Xmx2048m 
16 -Xmn128m
17 -Xss2m
18 -XX:PermSize=128m
19 -XX:MaxPermSize=256m
20 -XX:+UseParallelGC
21 -XX:+UseCompressedOops
22 -XX:+UseAdaptiveSizePolicy
23 -XX:+OptimizeStringConcat
24 -XstartOnFirstThread
25 -Dorg.eclipse.swt.internal.carbon.smallFonts
26 .....
```

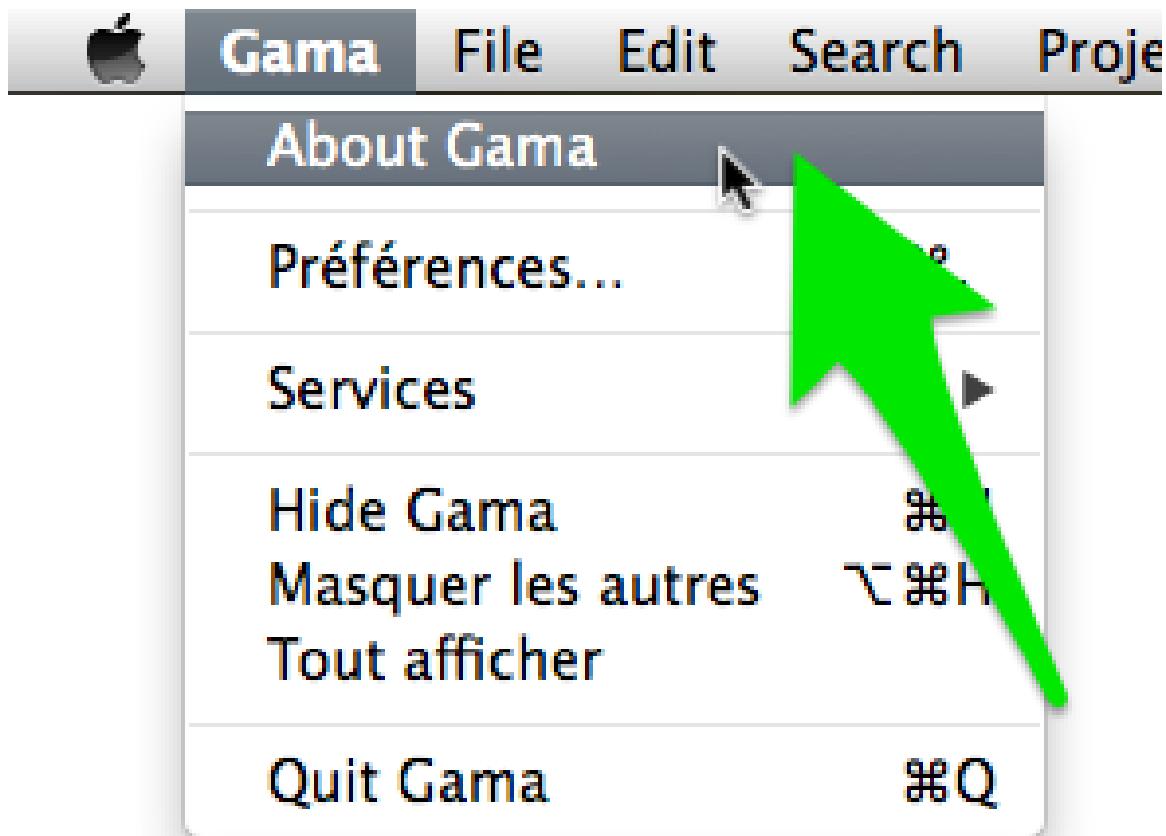
-Xms supplies the minimal amount of memory the JVM should allocate to GAMA, -Xmx the maximal amount. By changing these values (esp. the second one, of course, for example to 4096M !), saving the file and relaunching GAMA, you can probably solve your problem. Note that 32 bits versions of GAMA will not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

Submitting an Issue

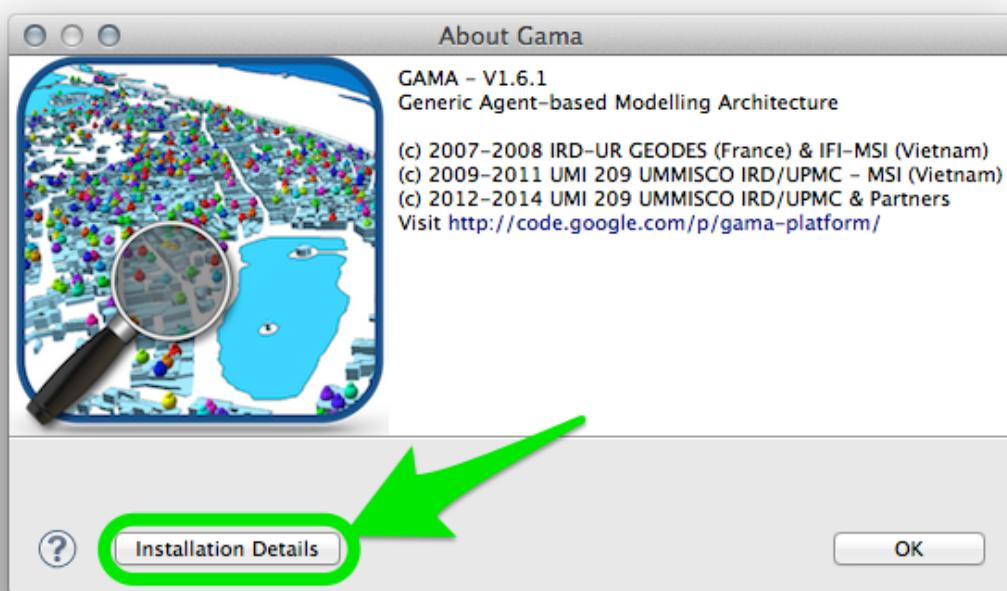
If you think you have found a new bug/issue in GAMA, it is time to create an issue report [here](#) ! Alternatively, you can click the [Issues](#) tab on the project site, search if a similar problem has already been reported (and, maybe, be solved) and, if not, enter a new issue with as much information 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.

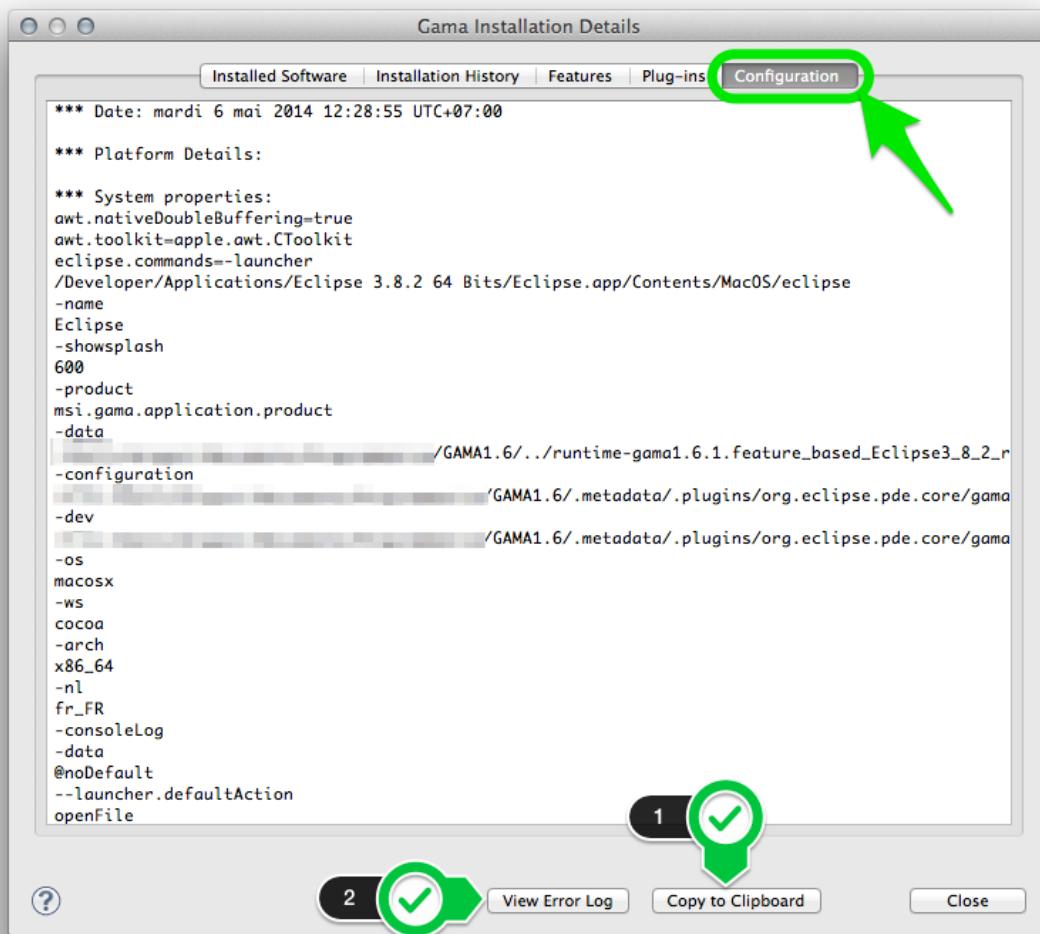
Two files may be particularly interesting to attach to your issue: the **configuration details** and the **error log**. Both can be obtained quite easily from within GAMA itself in a few steps. First, click the "About GAMA..." menu item (under the "Gama" menu on MacOS X, "Help" menu on Linux & Windows)



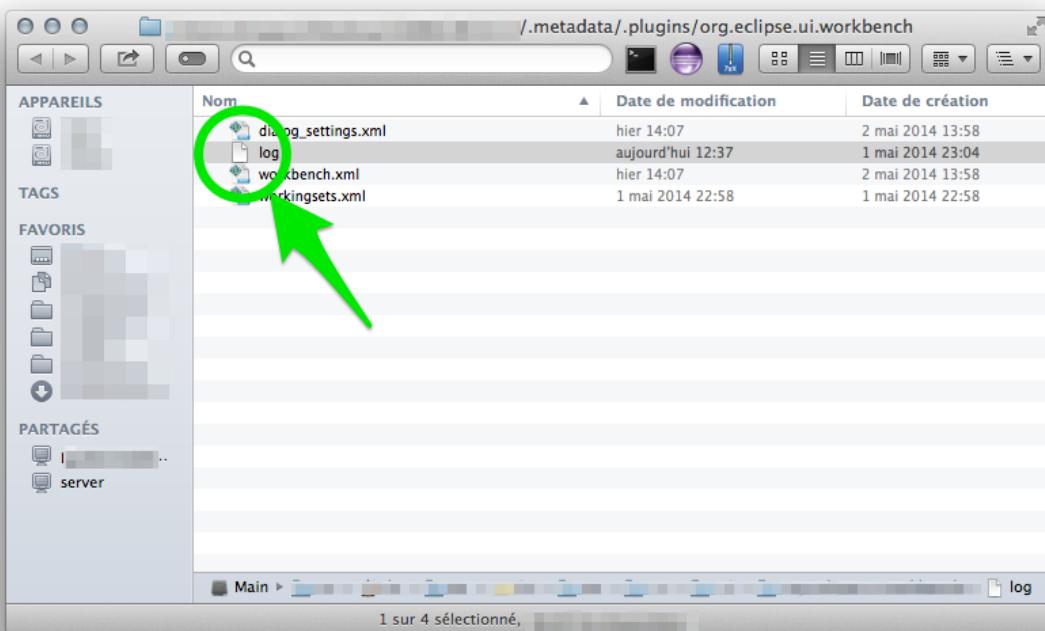
In the dialog that appears, you will find a button called "Installation Details".



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.



Please also visit the [Issue Tracker Help](#) for background information on how to write and compose issues.

2. Workspace, Projects and Models

Workspace, Projects and Models

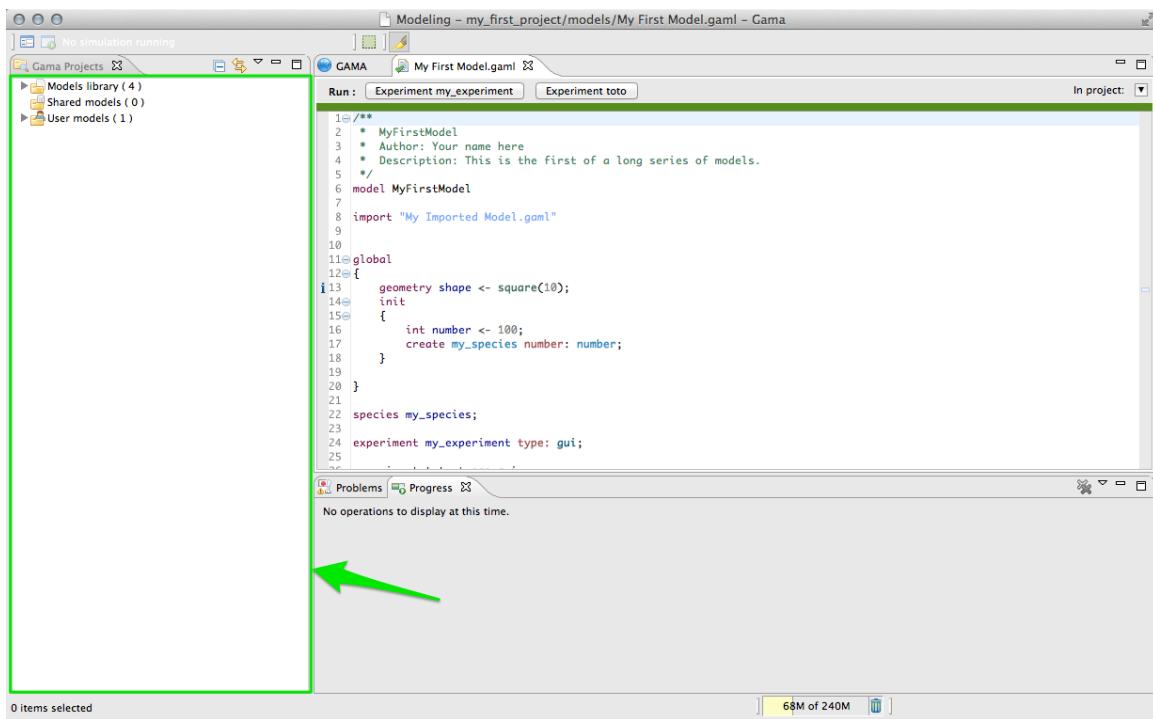
The **workspace** is a directory in which GAMA stores all the current projects on which the user is working, links to other projects, as well as some meta-data like preference settings, current status of the different projects, **error markers**, and so on. Except when running in **headless mode**, **GAMA 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, how to **import**, **export** or **share** models 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](#)
- 4. [Sharing Models](#)

2.1 Navigating in the Workspace

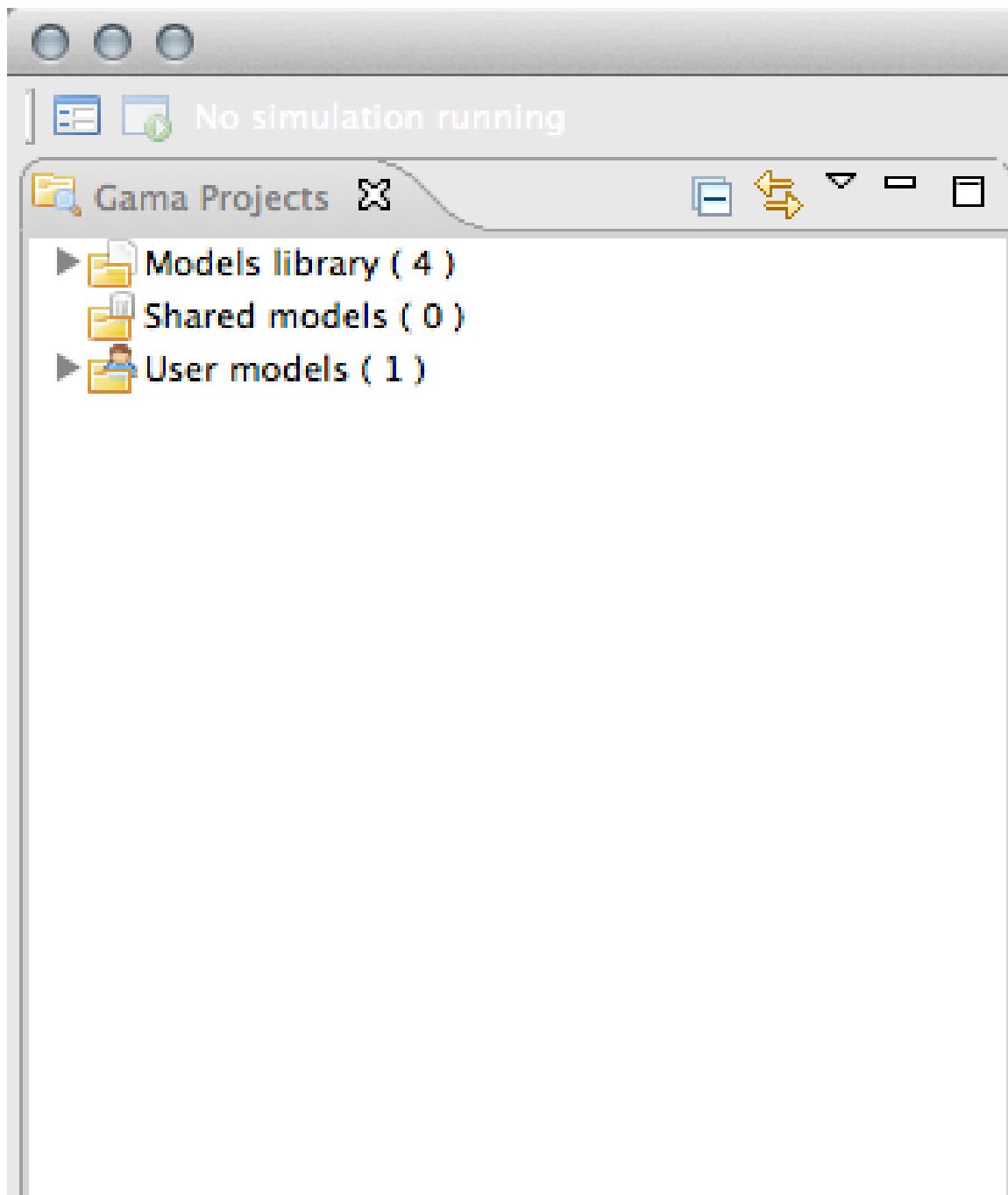
Navigating in the Workspace

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**.



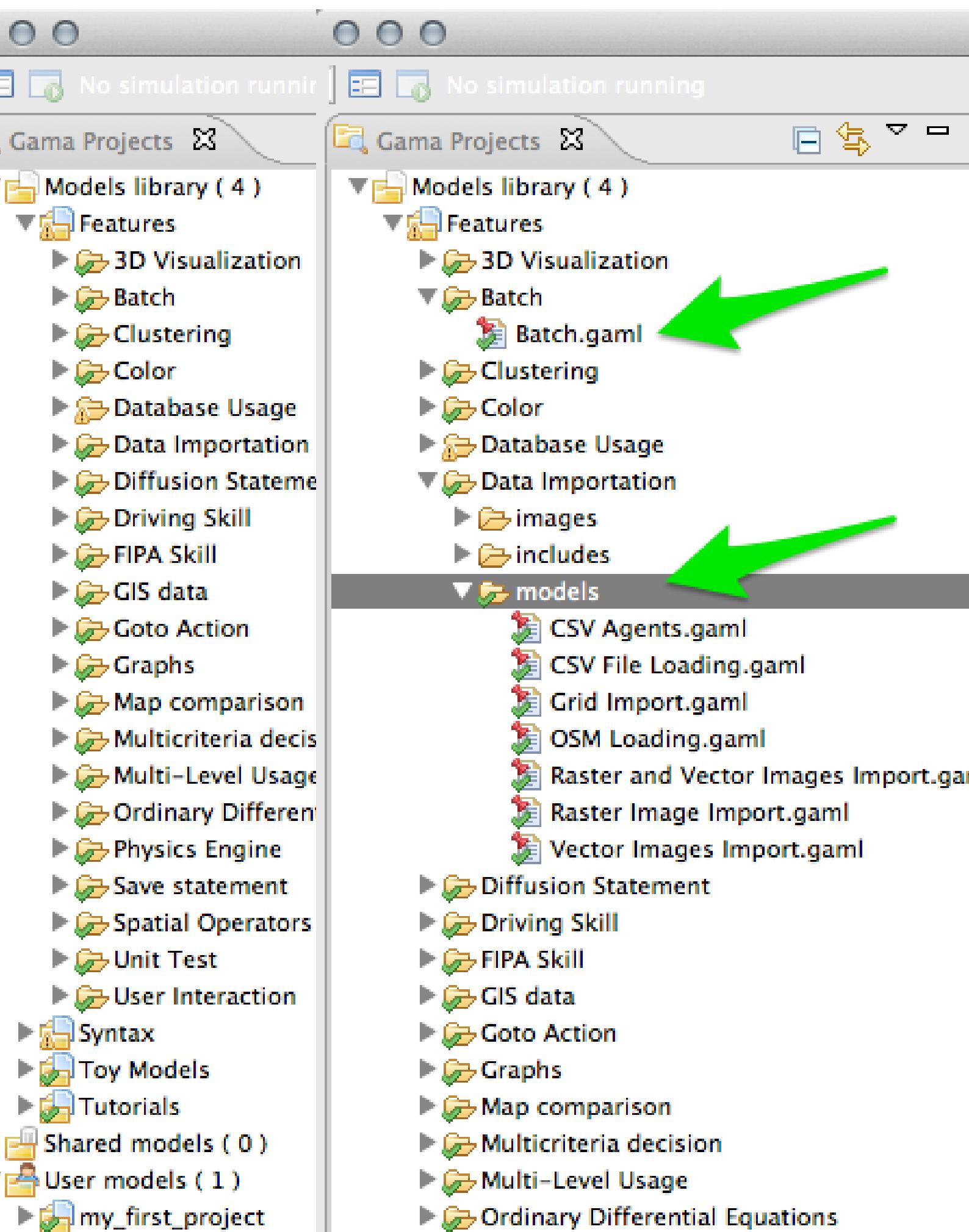
The Different Categories of Models

In the *Navigator*, models are organized in three different categories: *Models library*, *Shared 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.

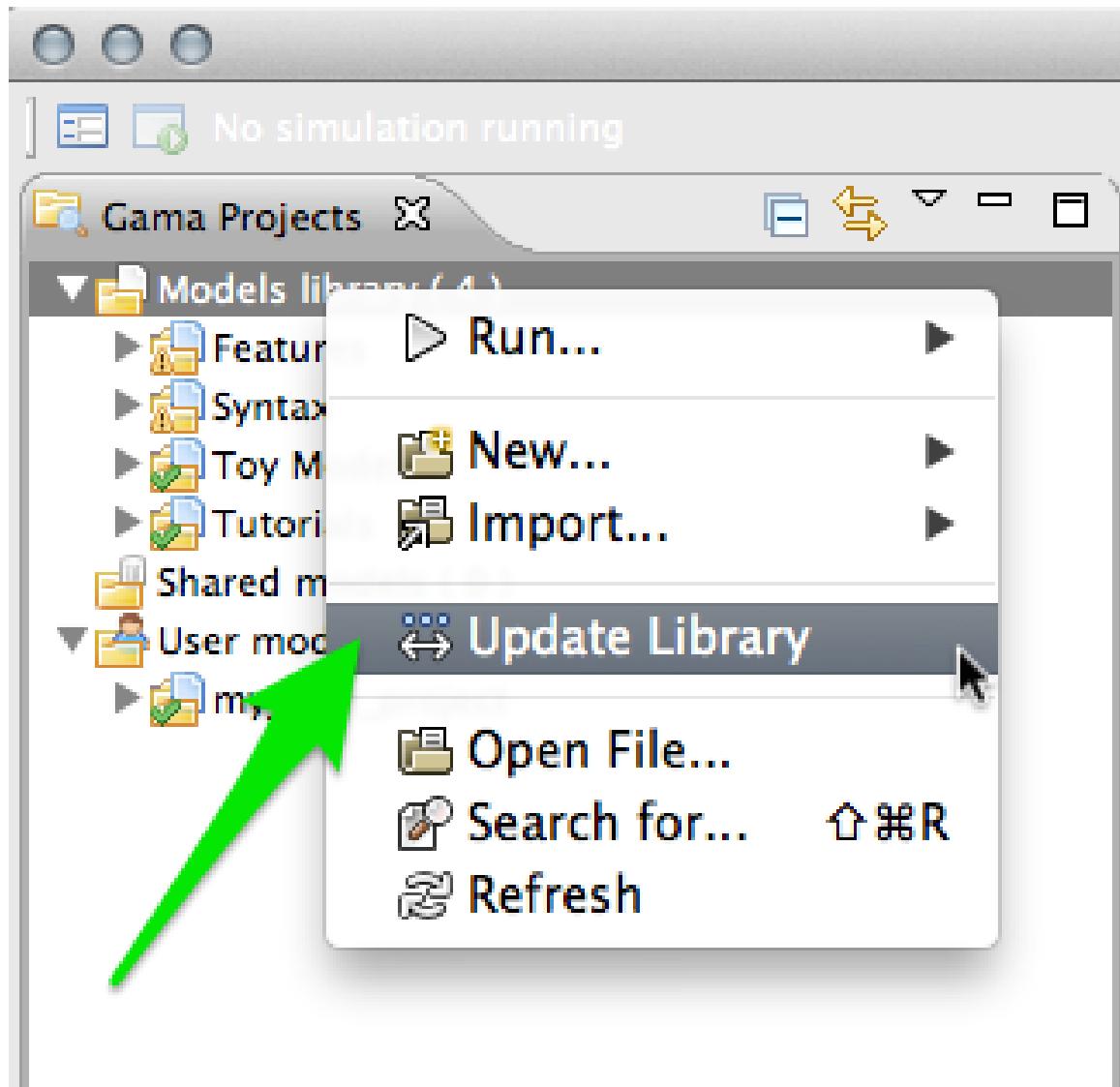


Models library

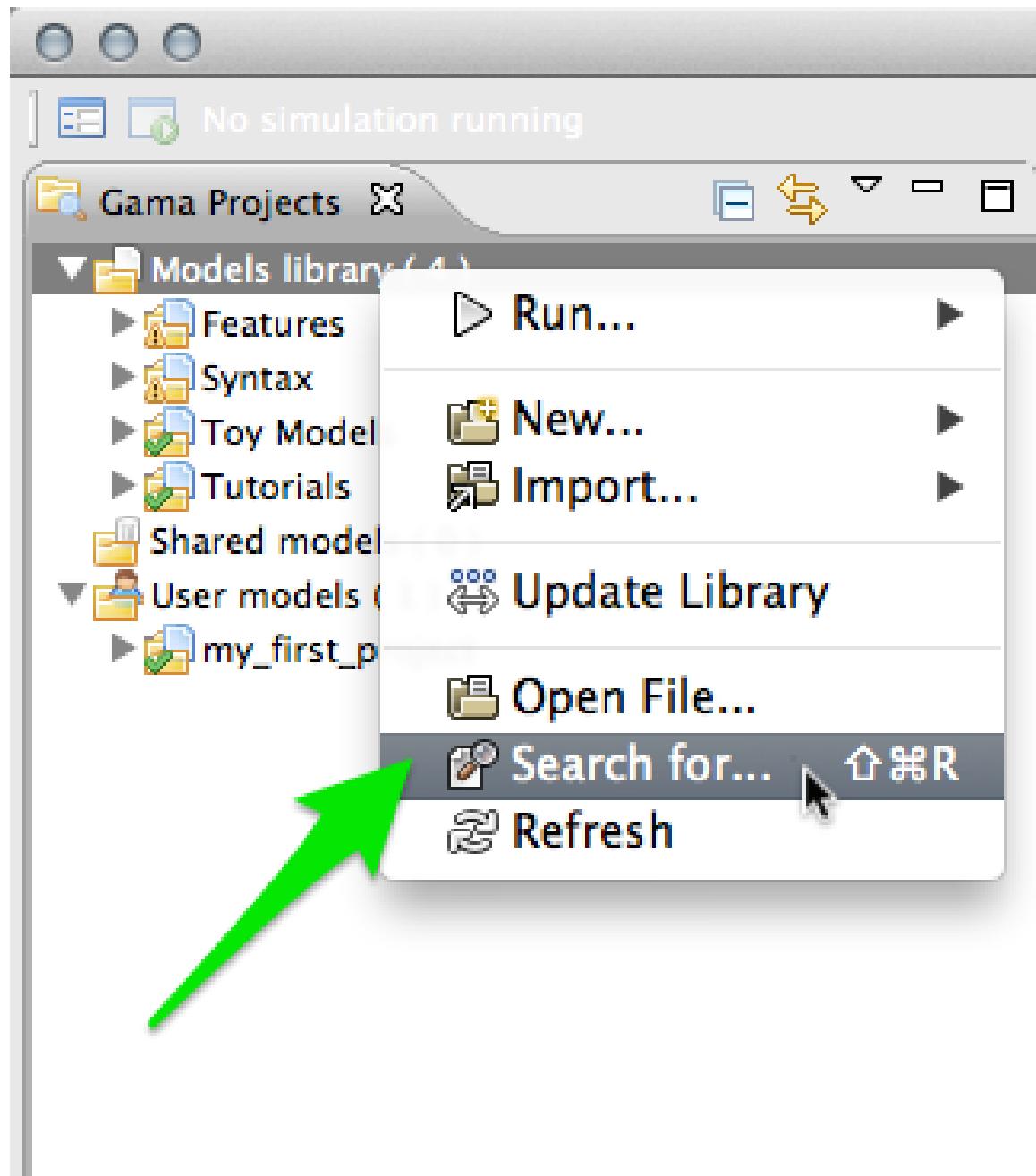
This category represents the models that are shipped with each version of GAMA. They do not reside in the workspace, but are considered as *linked* from it. This link is established every time a new workspace is created. Their actual location is within a plugin (msi.gama.models) of the GAMA application. 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.

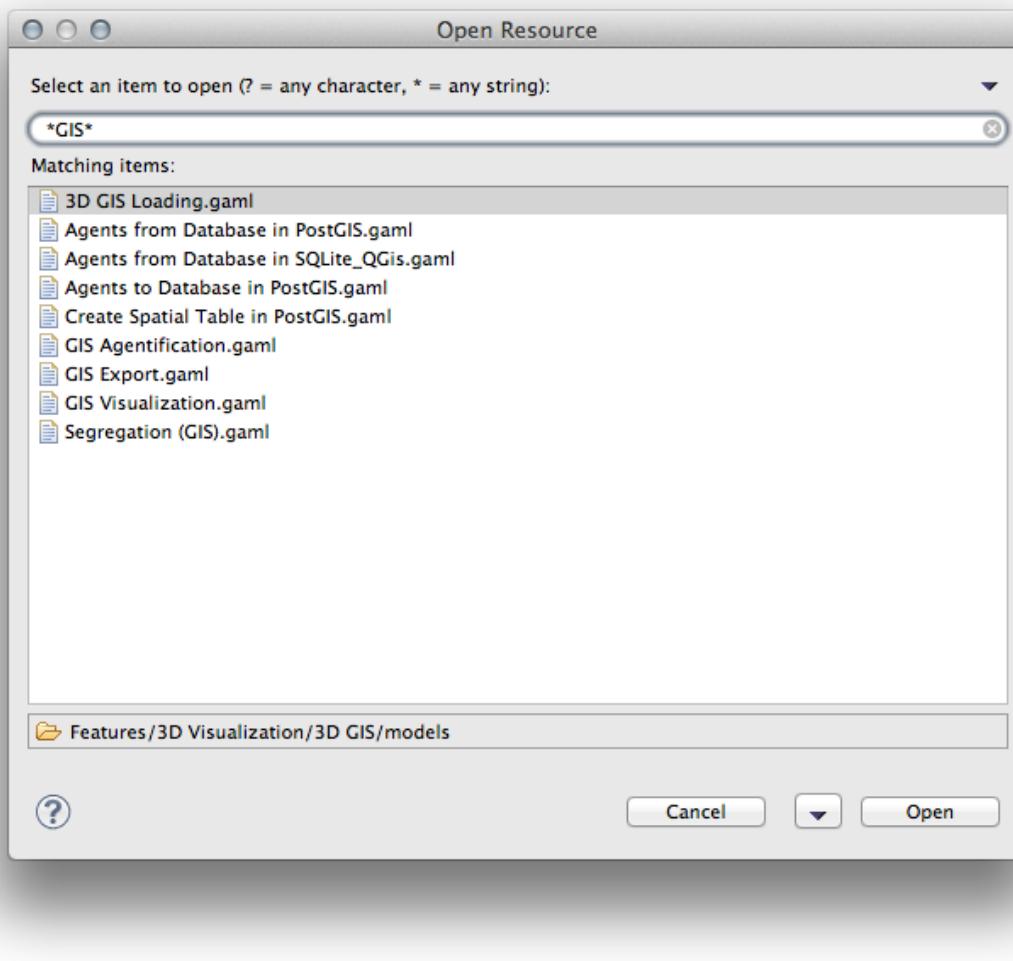


It may happen, in some occasions, that the library of models is not synchronized with the version of GAMA that uses your workspace. This is the case if you use different versions of GAMA to work with the same workspace. In that case, it is required that the library be manually updated. This can be done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



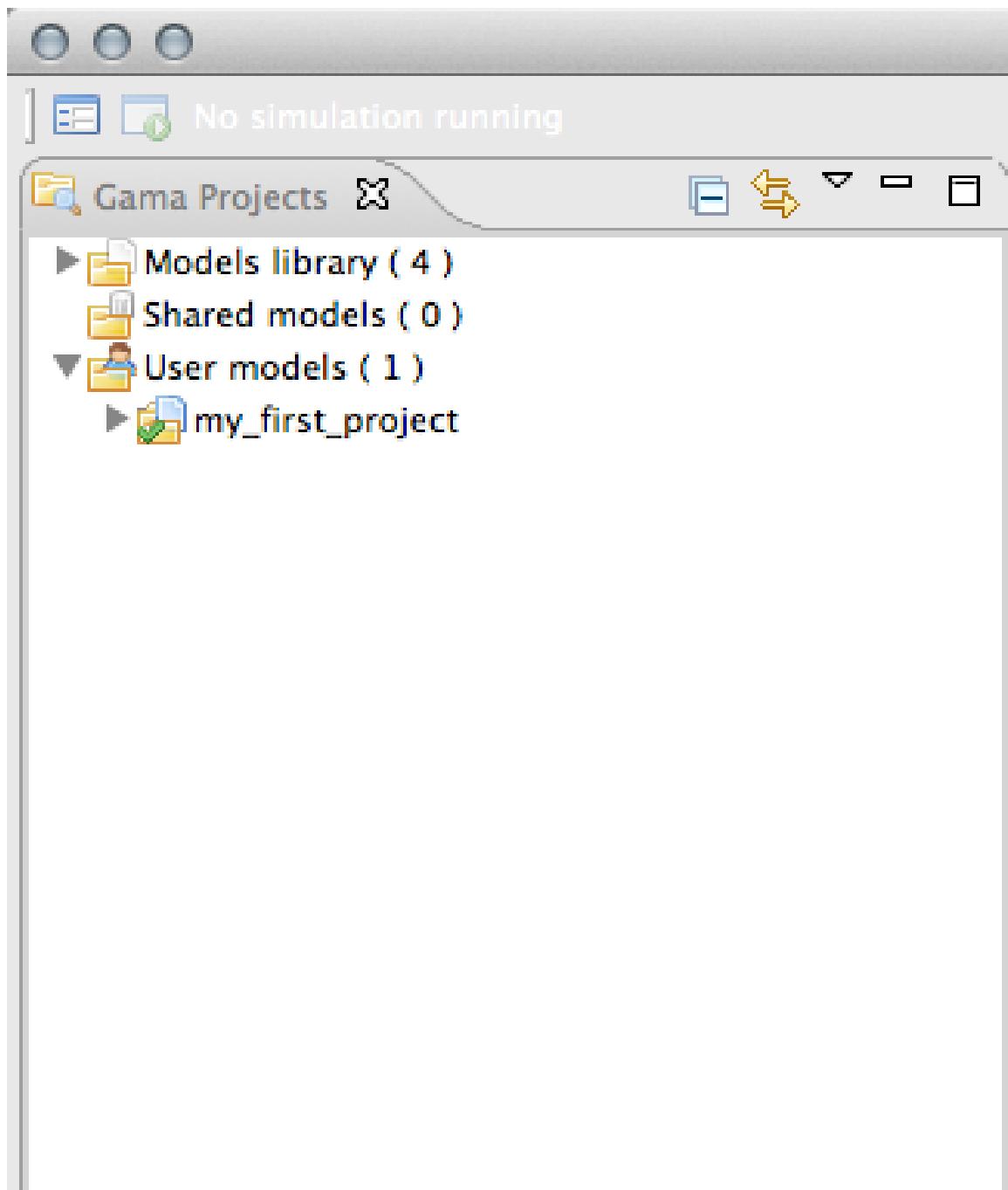
To look up for a particular model in the library, users can use the "Search for..." menu item. A search dialog is then displayed, which allows to look for models by their title (for example, models containing "GIS" in the example below).





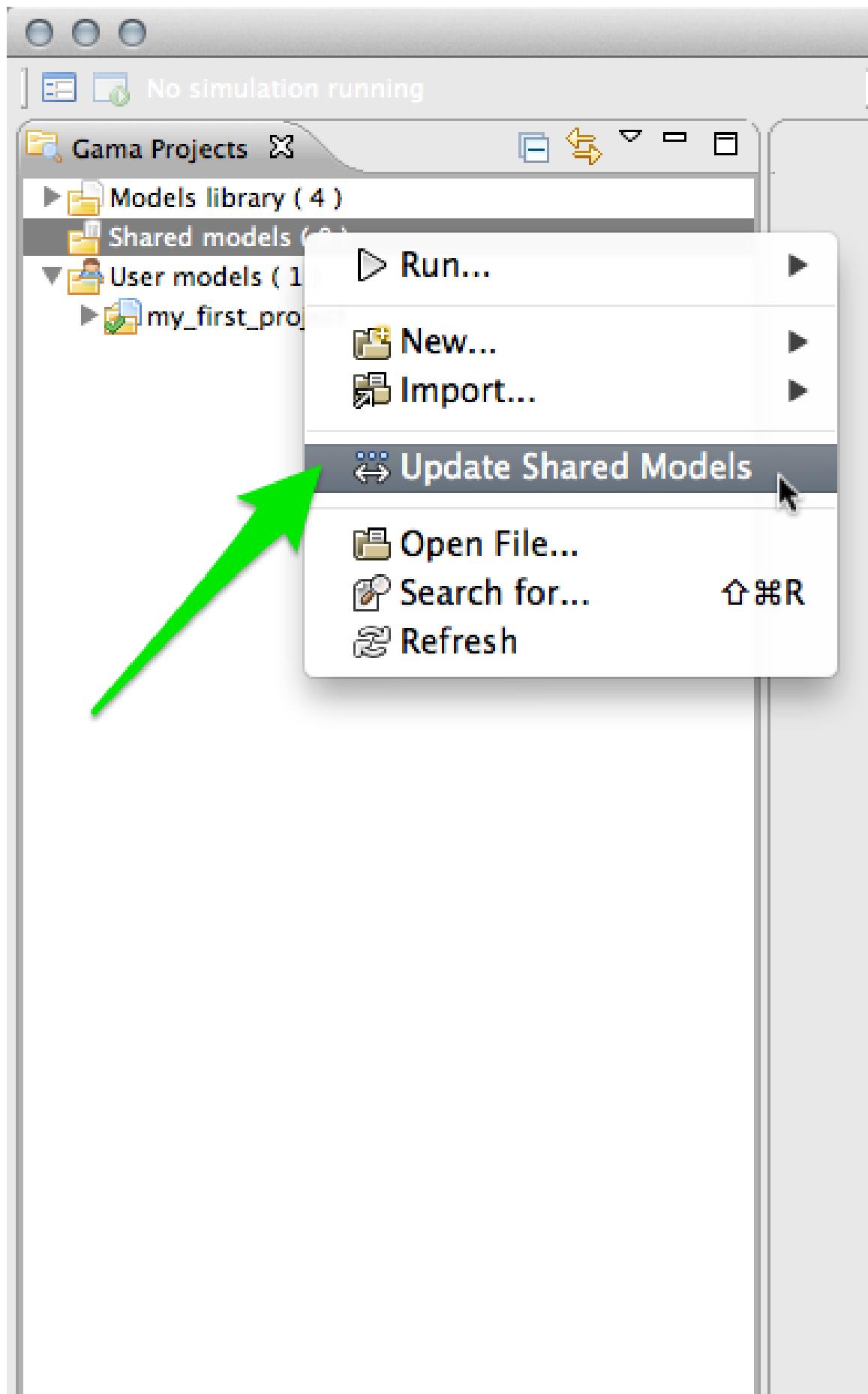
User models

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".

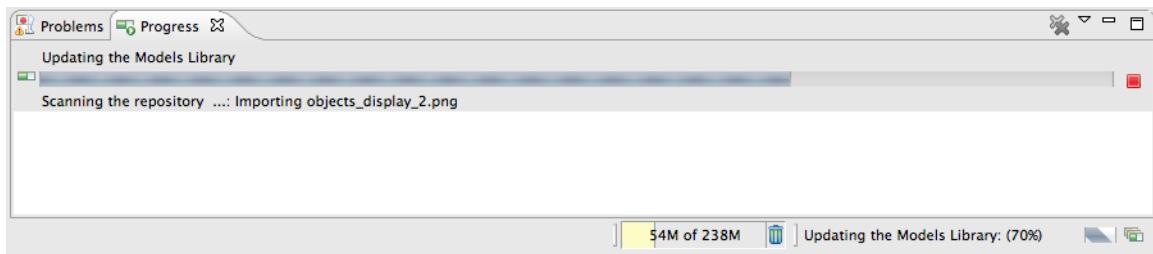


Shared models

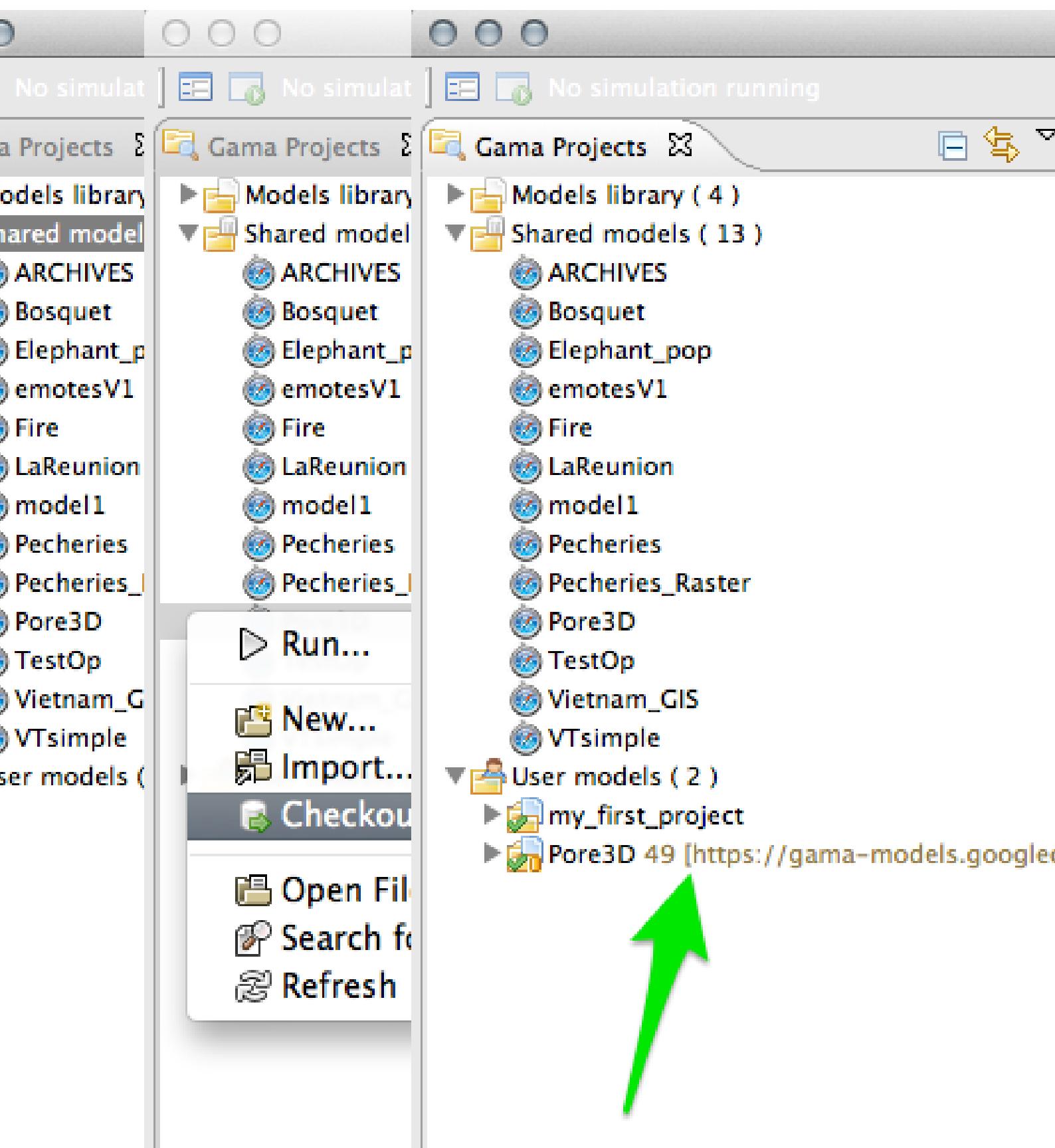
Shared projects are projects that reside in the SVN repository of the site gama-models.googlecode.com, which is intended to support GAMA users in easily sharing their models, as the access is built-in inside the platform. This category gives access to reading the models already contributed there (see [G_SharingModels](#) to see how to contribute). It is initially empty, but invoking the "Update Shared Models" menu item will make GAMA retrieve the list of projects present on the site.



This retrieval is implemented to be as light as possible so as not to lock up the platform for a long time while the SVN client is reading up the information. It happens in the background, and only retrieves the description of models (when it is available).



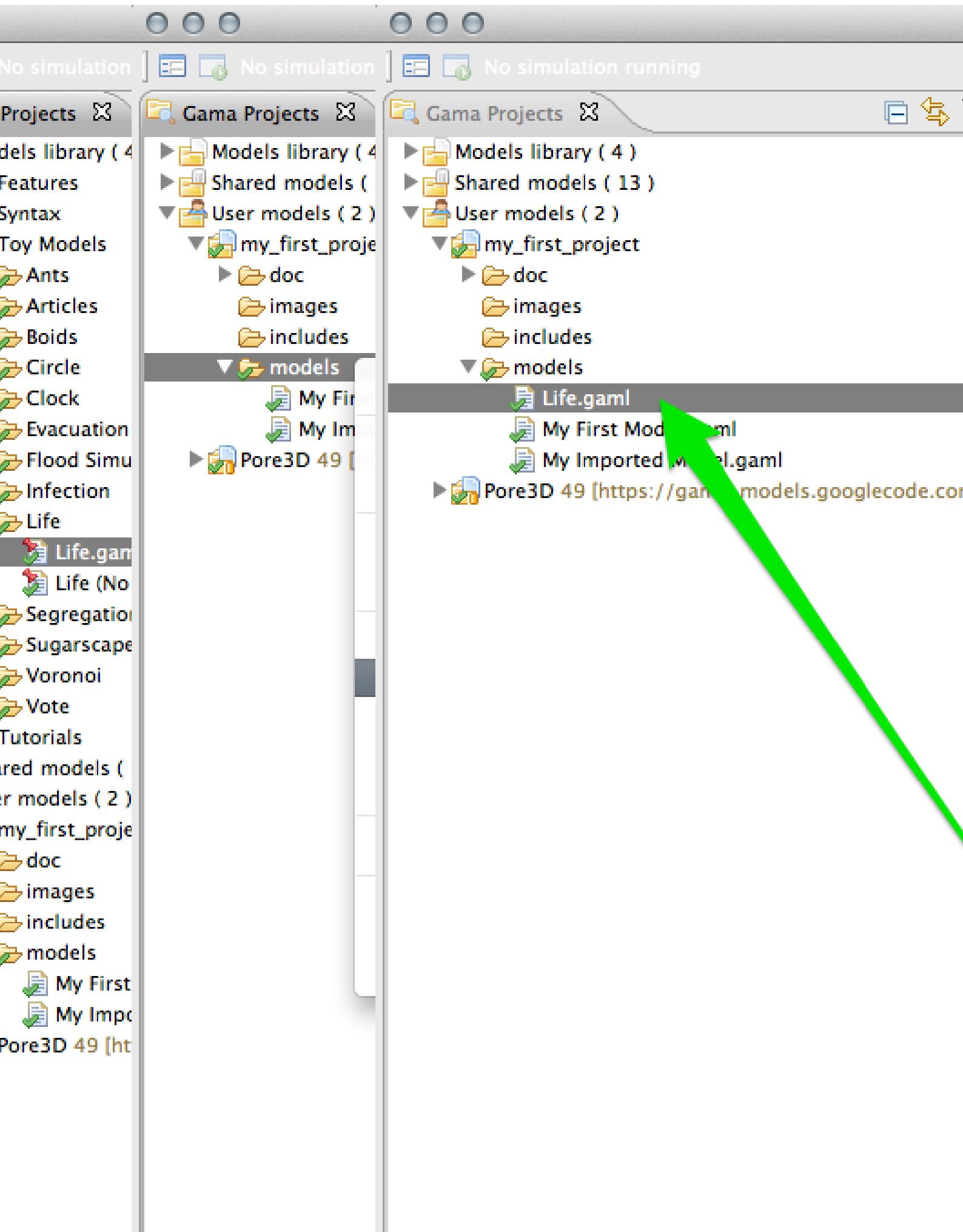
When the process is done, the navigator will display the html descriptions of the different projects found in the repository (the list below is indicative of the state of the SVN server as of May 2014, but the actual list of projects will undergo several changes as most of them are not compatible with the latest version of GAMA). To retrieve a complete project, the user has to choose its corresponding description file and invoke "Checkout Project from Repository" from the contextual menu. After the download is over, the project will appear in the "User Models" category, sporting information on its revision number and location to distinguish it from the "regular" models already present in this category.



If the user has a write access to the gama-models SVN repository, any modification he/she makes to the model can be committed back to the repository using the standard "Team..." submenu (see [G_SharingModels](#)) and subsequently shared with all GAMA users.

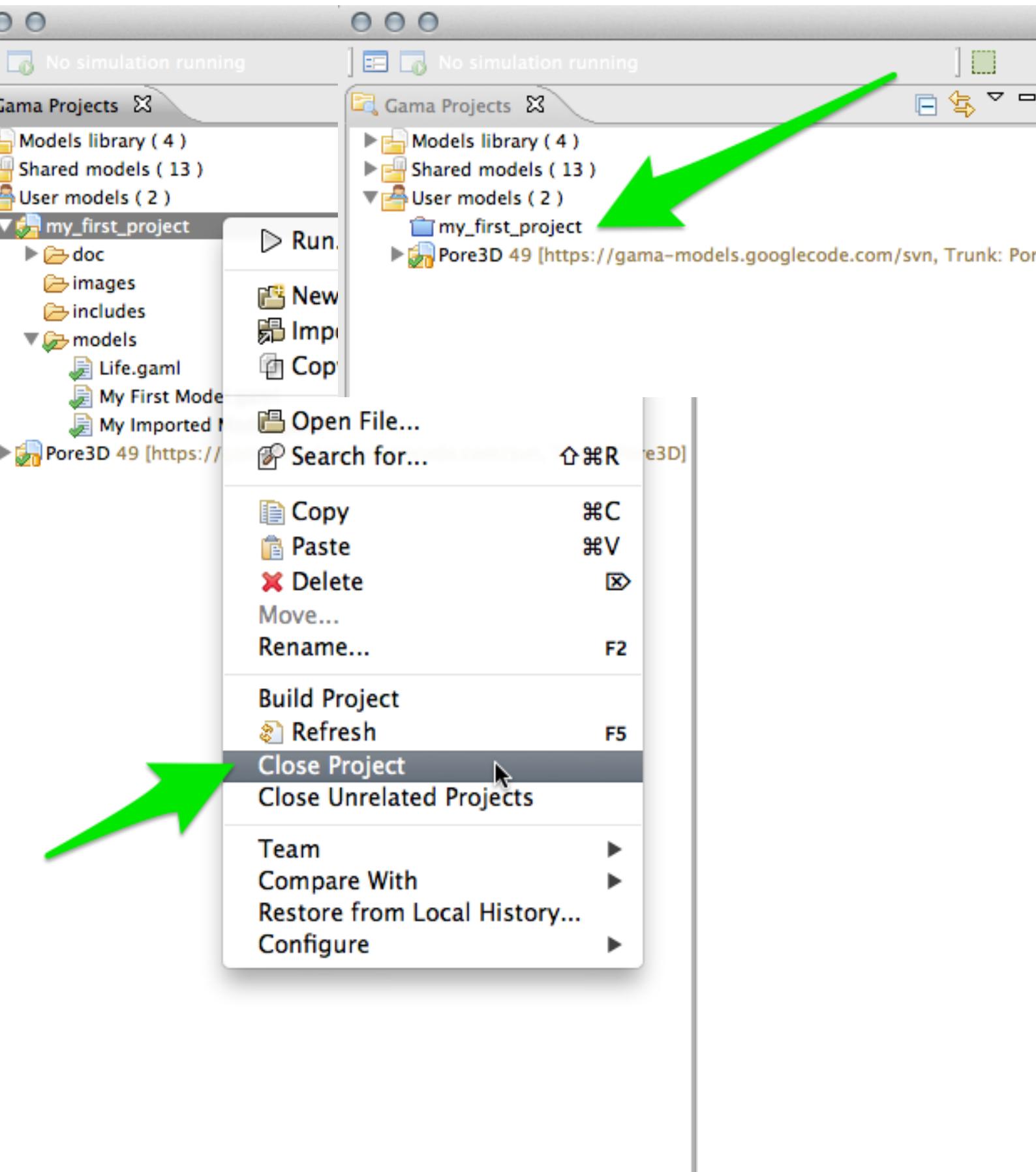
Moving Models Around

Model files, as well as resources, or even complete projects, can be moved around between the "Models Library" 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.

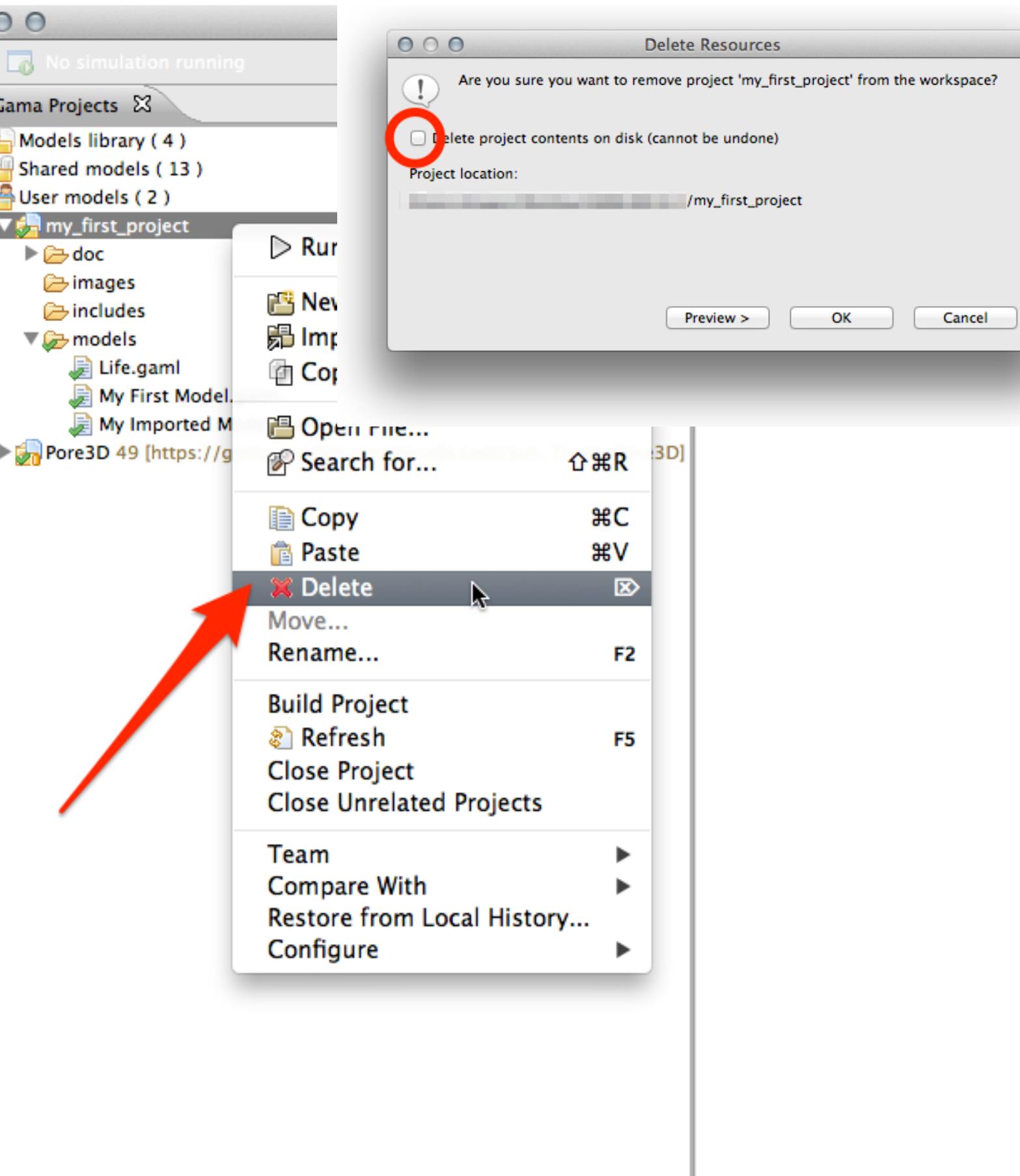


Closing and Deleting Projects

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.



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.



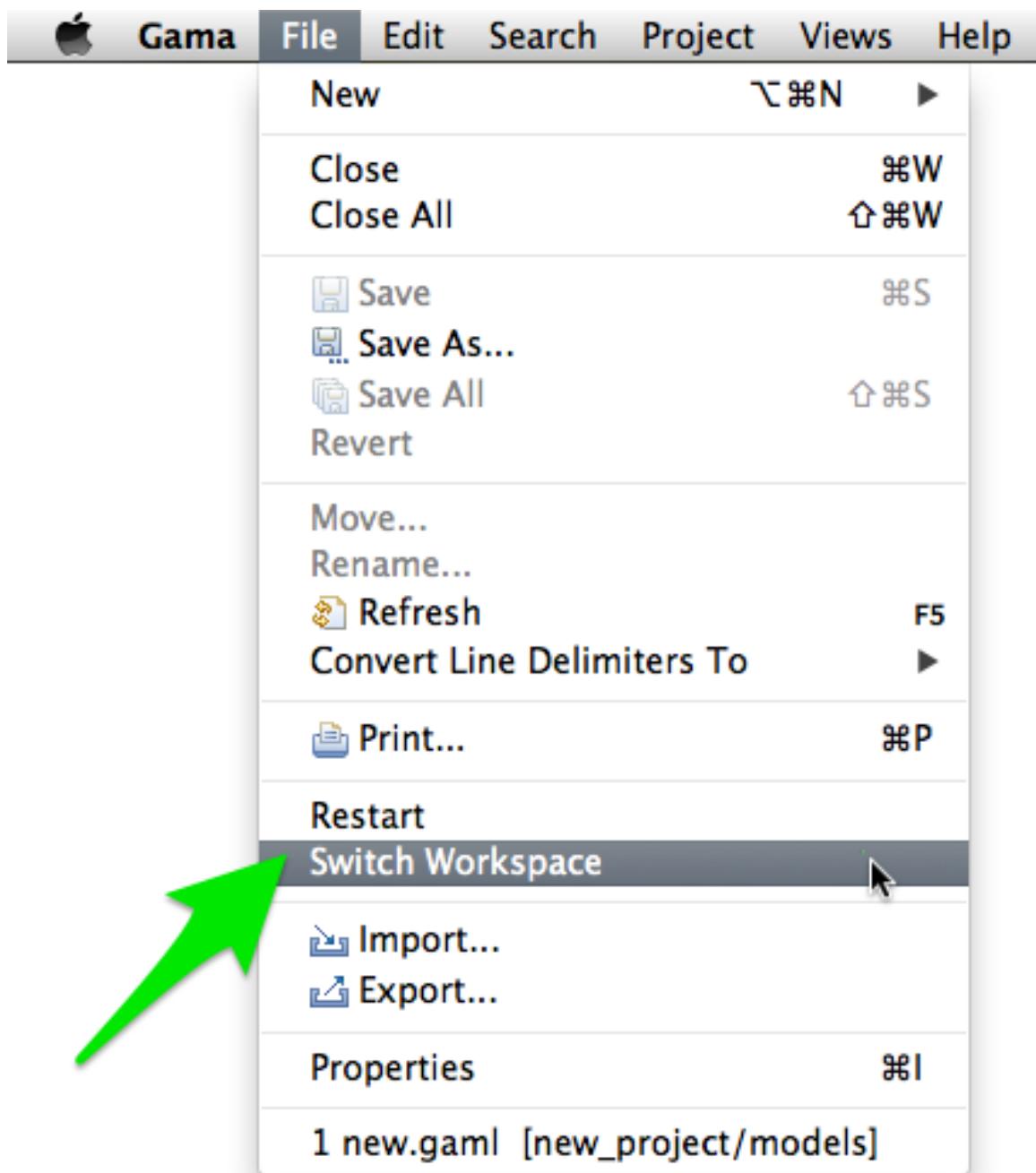
2.2 Changing Workspace

Changing Workspace

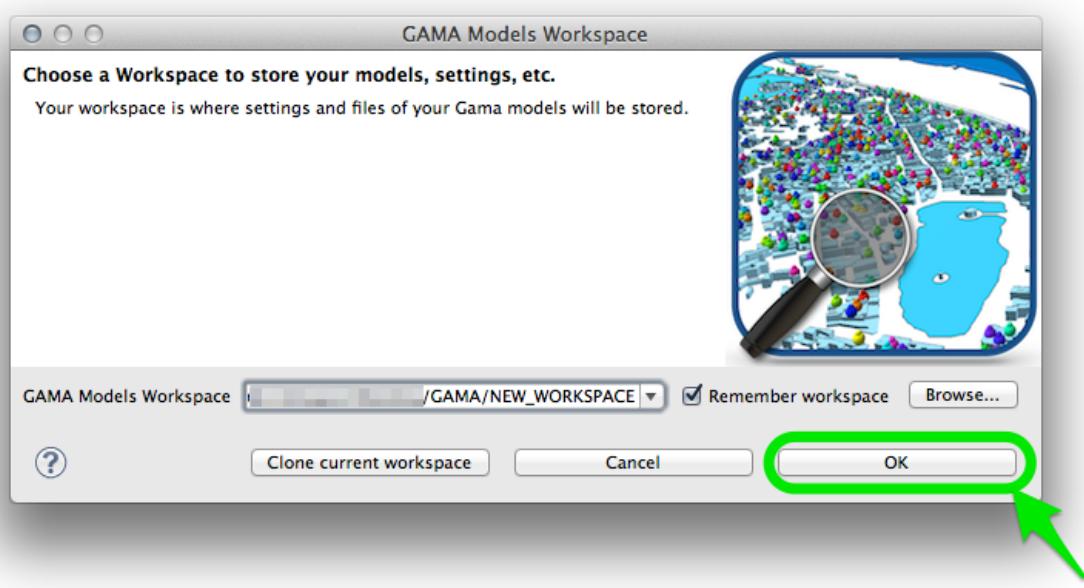
—
It is possible, and actually common, to store different projects/models in different workspaces and to tell GAMA to switch between these workspaces. Doing so involves being able to create one or several new workspace locations (even if GAMA has been told to [remember](#) the current one) and being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

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.

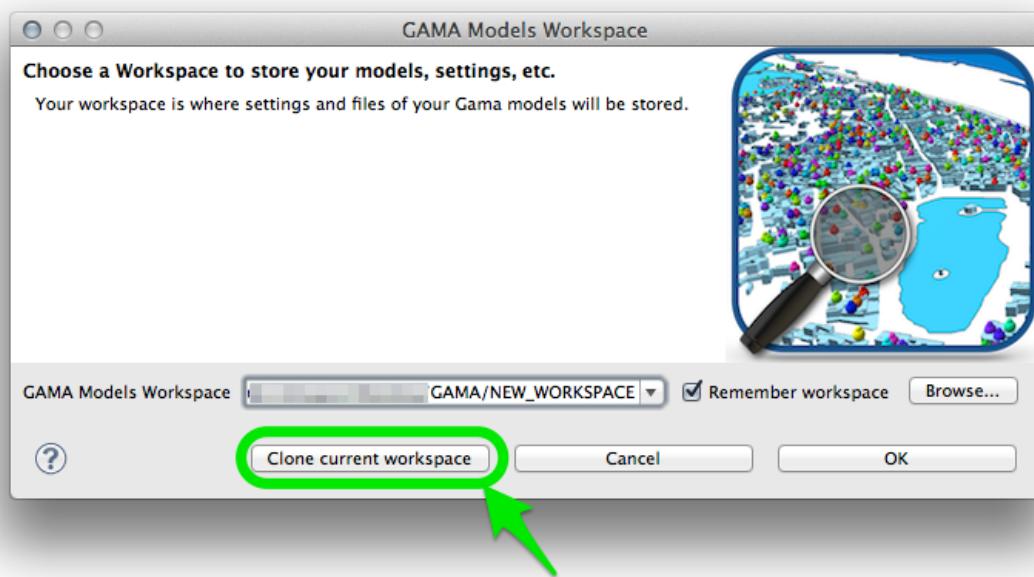


In the dialog that appears, the current workspace location should already be entered. Changing it to a new location (or choosing one in the file selector invoked by clicking on "Browse...") and pressing "OK" will then either create a new workspace if none existed at that location or switch to this new workspace. Both operations will restart GAMA and set the new workspace location. To come back to the previous location, just repeat this step (the previous location is normally now accessible from the combo box).

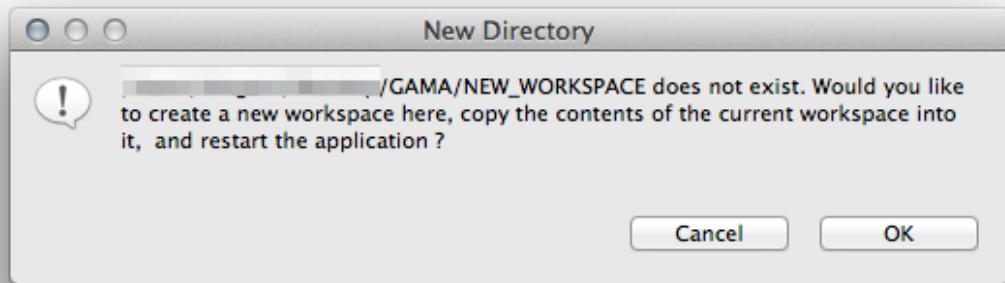


Cloning the Current Workspace

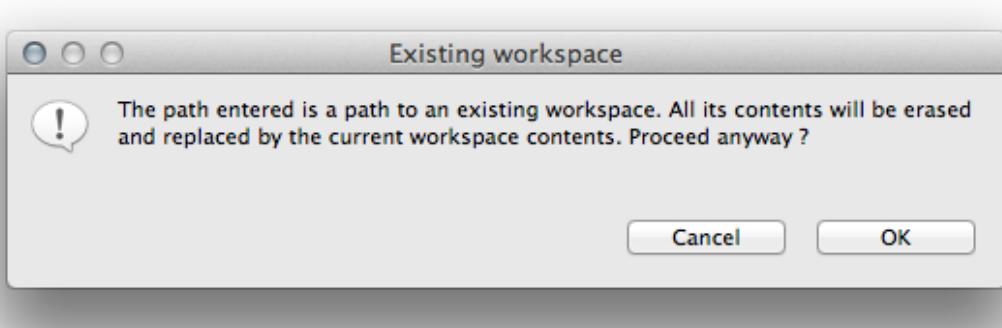
Another possibility, if you have models in your current workspace that you would like to keep in the new one (and that you do not want to [import](#) one by one after switching workspace), or if you change workspace because you suspect the current one is corrupted, or outdated, etc. but you still want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



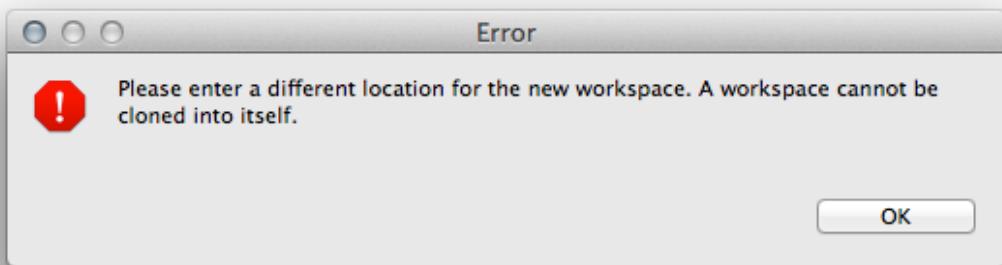
If the new location does not exist, GAMA will ask you to confirm the creation and cloning using a specific dialog box. Dismissing it will cancel the operation.



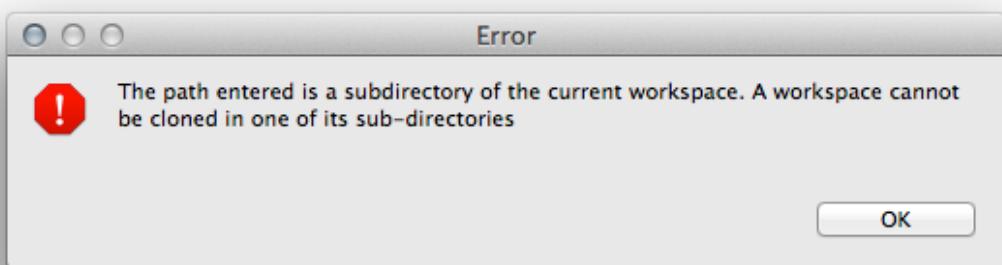
If the new location is already the location of an existing workspace, another confirmation dialog is produced. **It is important to note that all projects in the target workspace will be erased and replaced by the projects in the current workspace if you proceed**. Dismissing it will cancel the operation.



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).



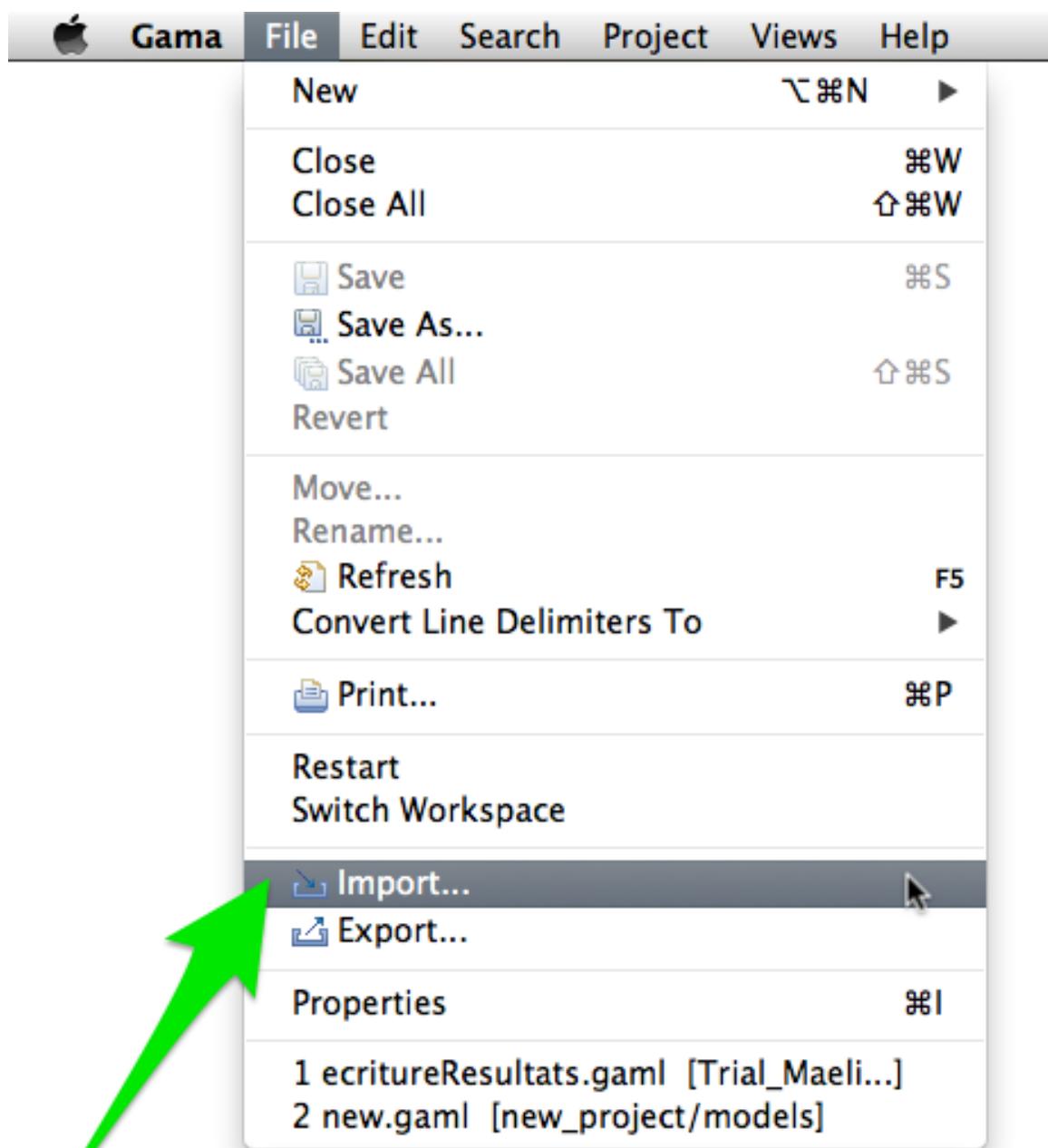
2.3 Importing Models

Importing Models

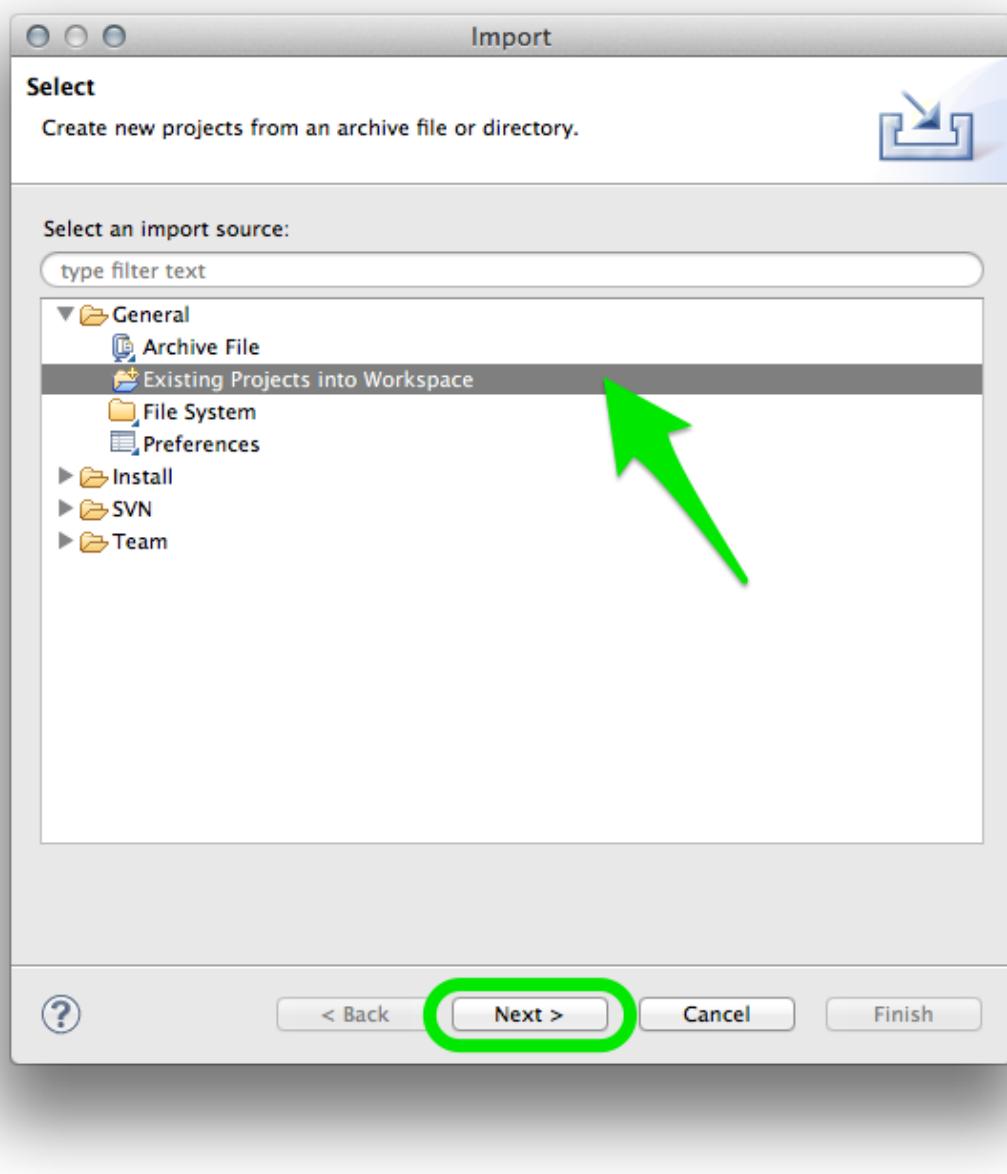
Importing a model refers to making a model file (or a complete project) available for edition and experimentation in the **workspace** . With the exception of **headless** experiments, GAMA requires that models be manageable in the current workspace to be able to validate them and eventually experiment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

The "Import..." Menu Command

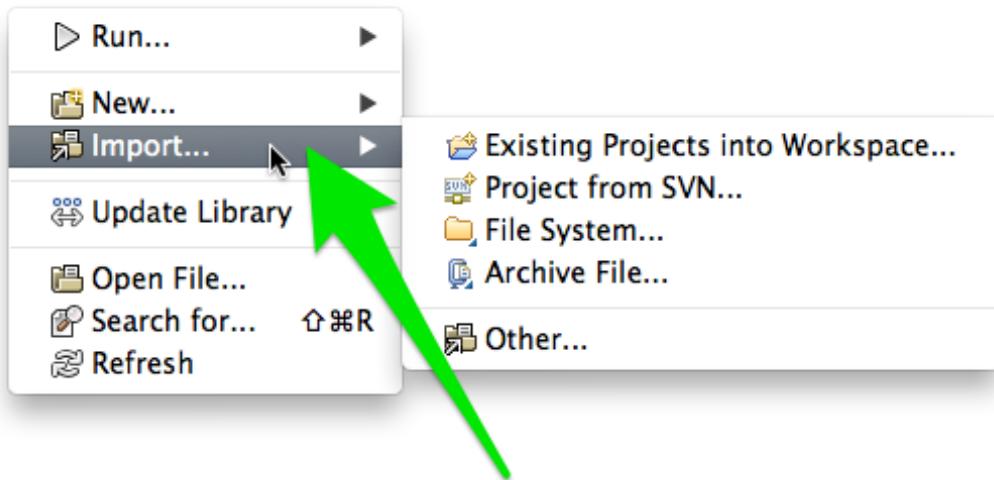
The simplest, safest and most secure way to import a project into the workspace is to use the built-in "Import..." menu command, available in the "File" menu or in the contextual menu of the *Navigator* .



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".

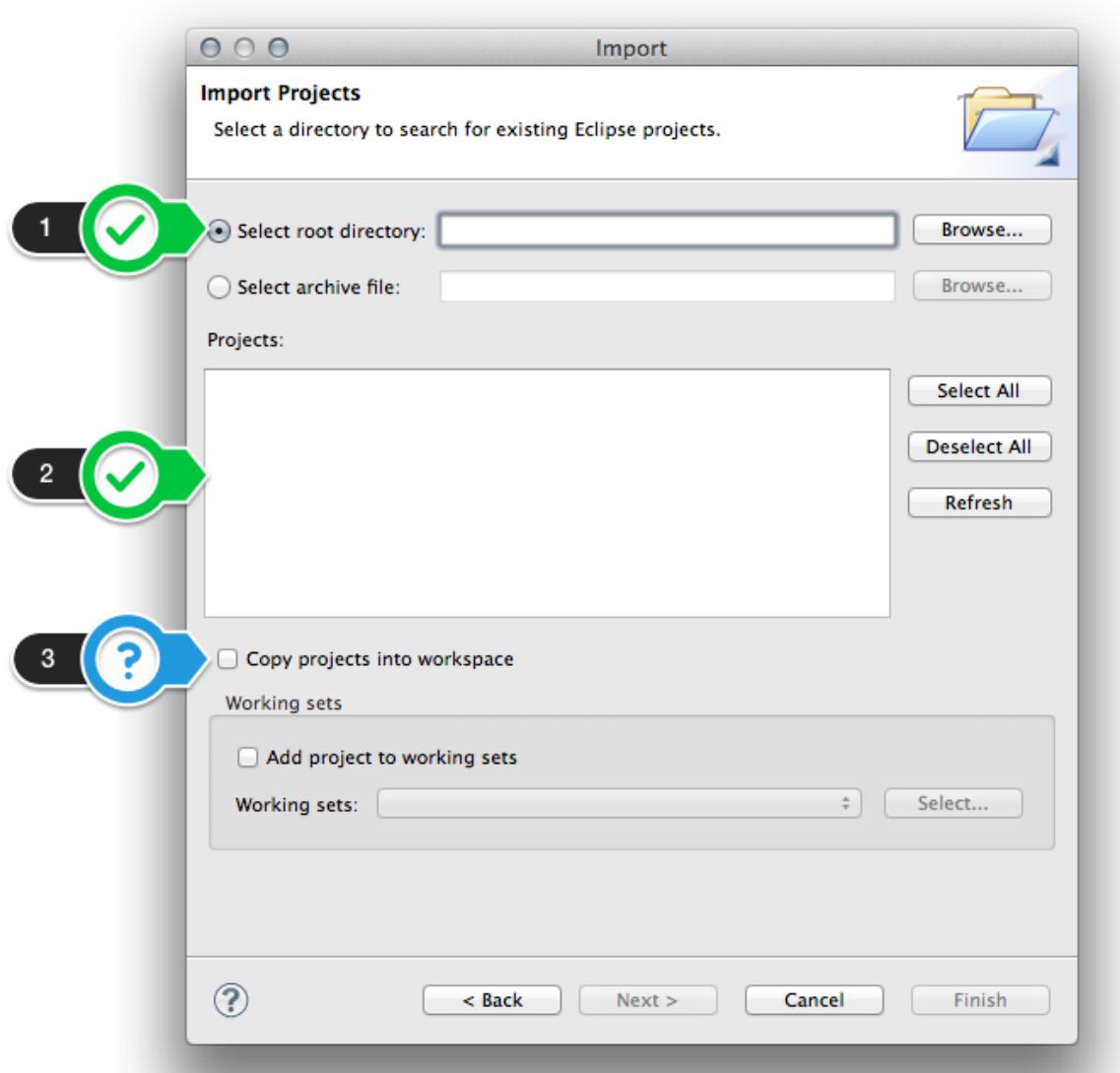


Note that when invoked from the contextual menu, "Import..." will directly give access to a shortcut of this source in a submenu.



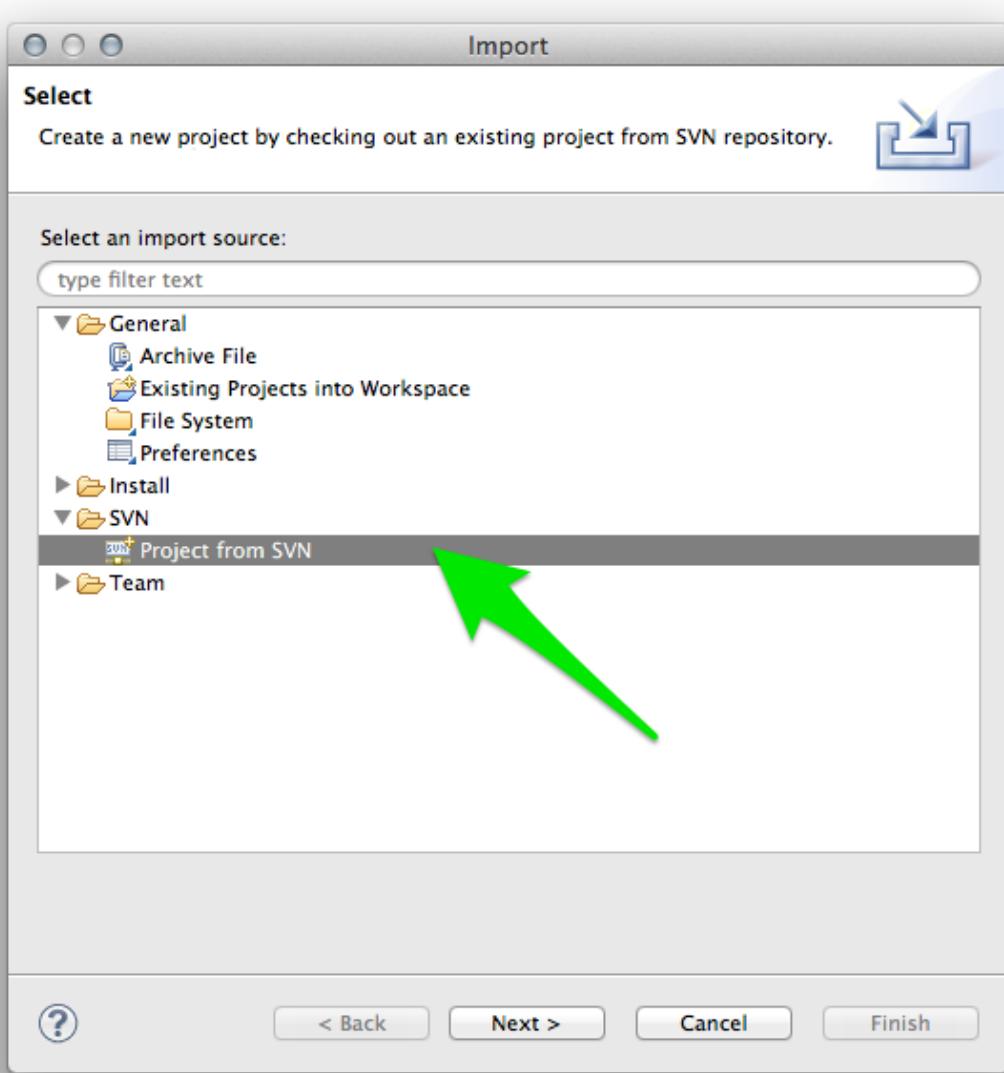
Both options will lead the user to a last dialog where he/she will be asked to:

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)

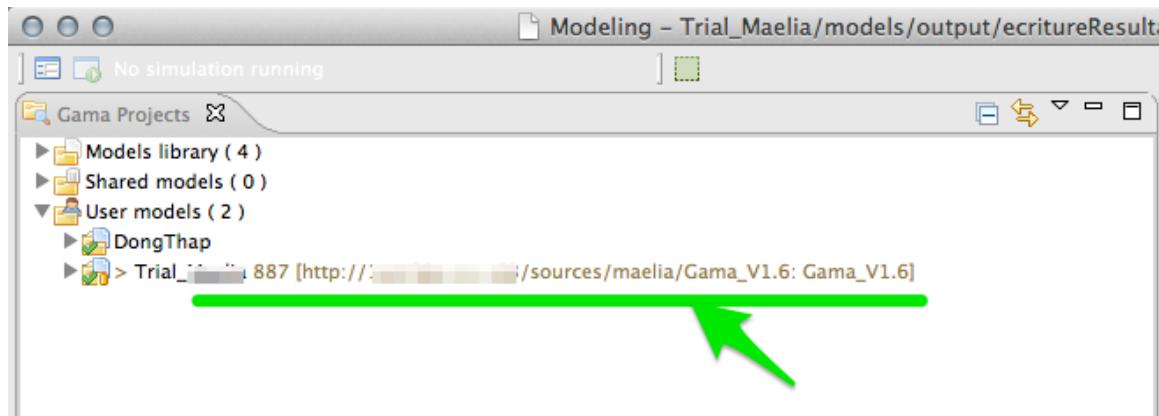


SVN import

If the user has access to a SVN repository containing GAMA projects, he/she can import them (i.e. *check them out* in SVN jargon) into the current workspace without problem. The option to choose in the previous dialog is "Projects from SVN".



The user is then asked for a SVN repository location, which may have been entered previously or not (in which case, he/she can create it from this dialog). GAMA, once the location is validated and accessible, will scan the repository for GAMA projects and propose them in a way similar to the previous one. Once imported from SVN, projects will remain "linked" with the repository they have been imported from (which allows to update them if the repository changes, or commit them back if the SVN is accessible on writing by the user). This "link" can be easily spotted in the *Navigator* because of the different label these projects will have:



Note that if the workspace is [cloned](#), these "links" to SVN repositories will remain untouched, allowing to have different workspaces pointing to the same SVN project.

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 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.

2.4 Sharing Models

Sharing Models

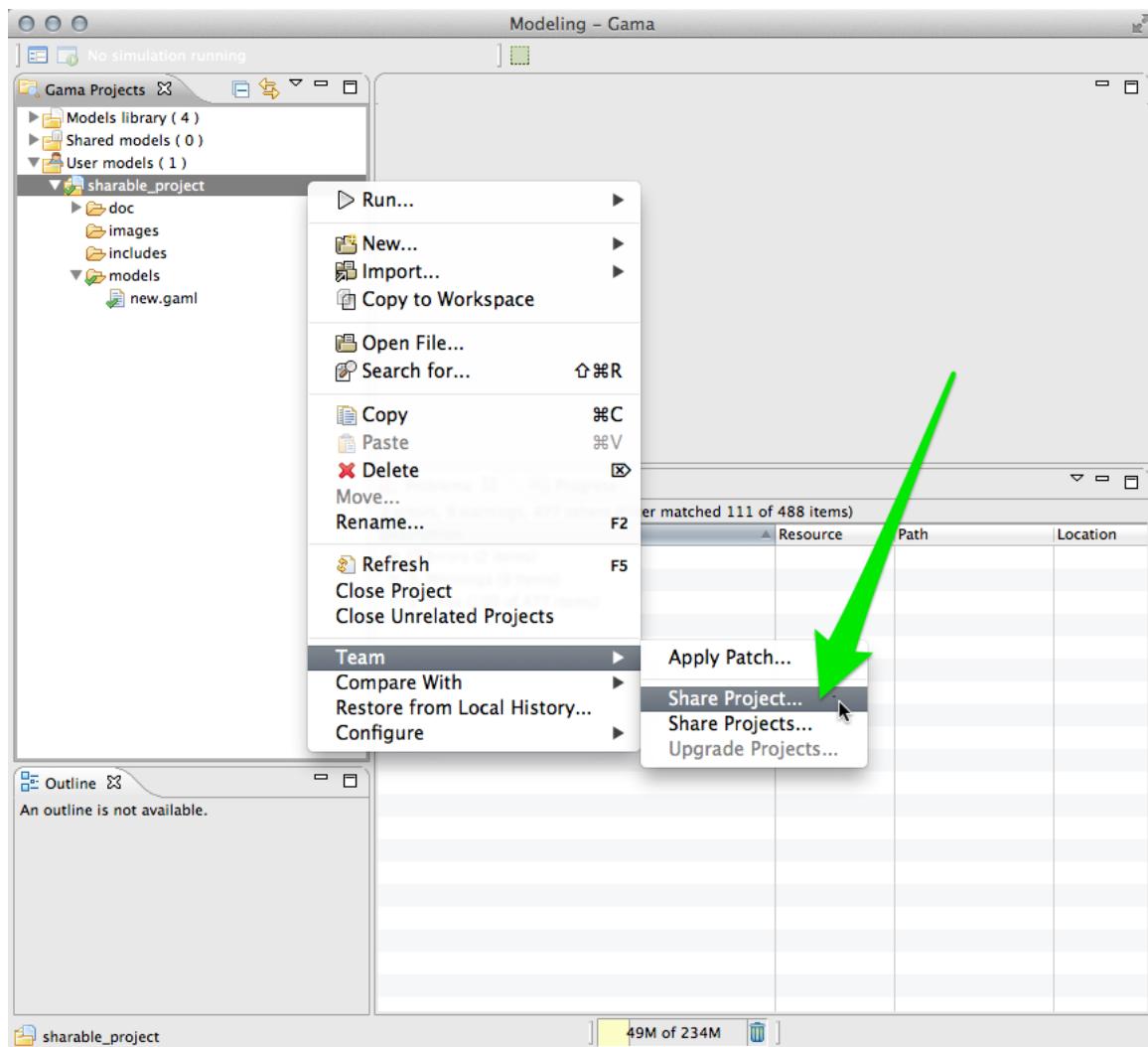
—

Thanks to its native integration of the SVN versioning system, GAMA offers users the possibility to share their projects (provided they have access to a SVN repository), without any further need to install add-ons or plugins. This section provides some basic information on how to share a project and manage it once it has been shared. For more specific documentation about SVN or other ways to use the built-in sharing facilities with other versioning systems, please refer to this [Eclipse documentation on SVN](#) and [this one on Git](#) (go to [this page](#) to learn how to install Git). Note that projects **imported** from a repository are automatically considered as "shared" between the workspace used by GAMA and the repository they have been imported from.

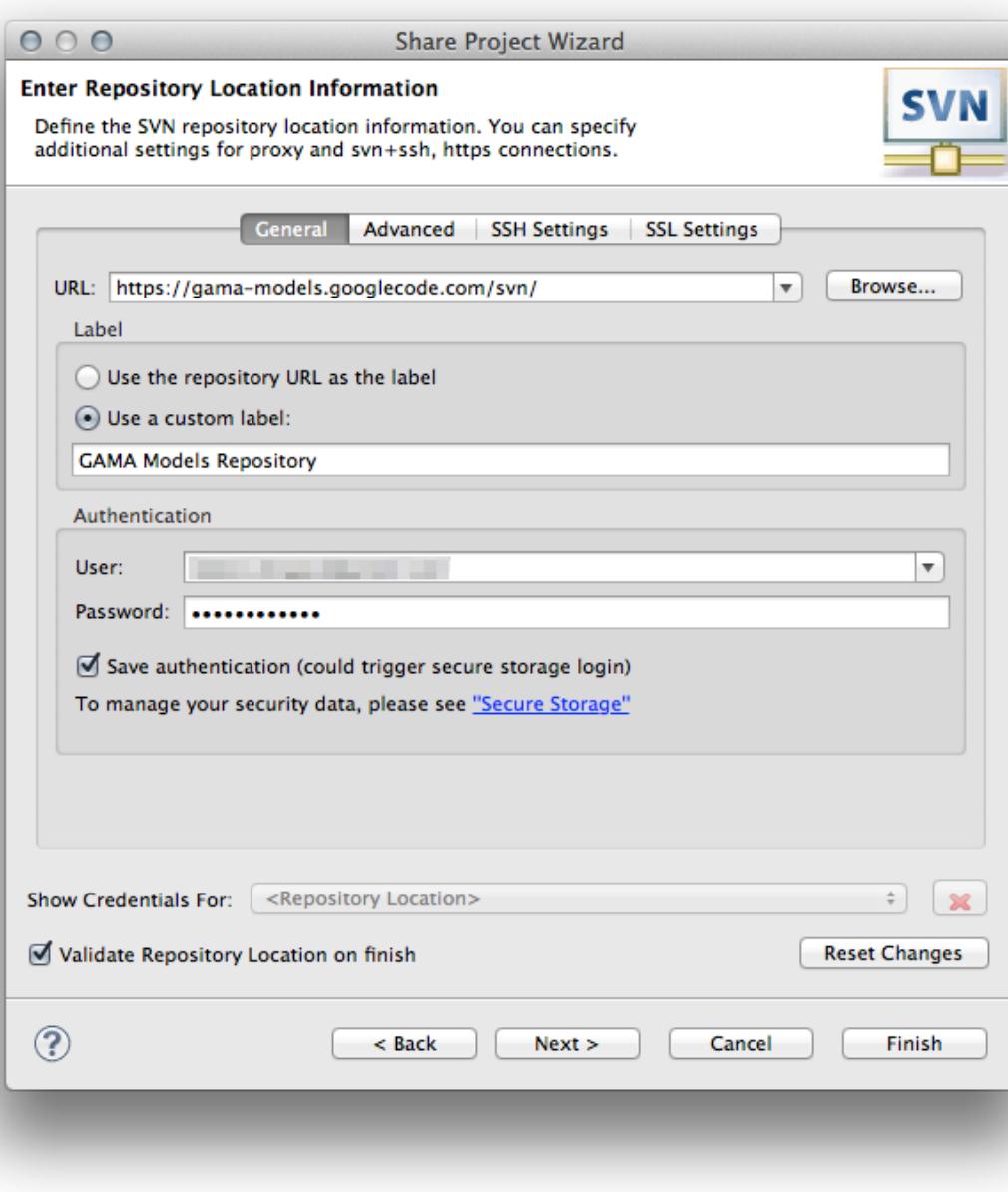
—

Project Sharing

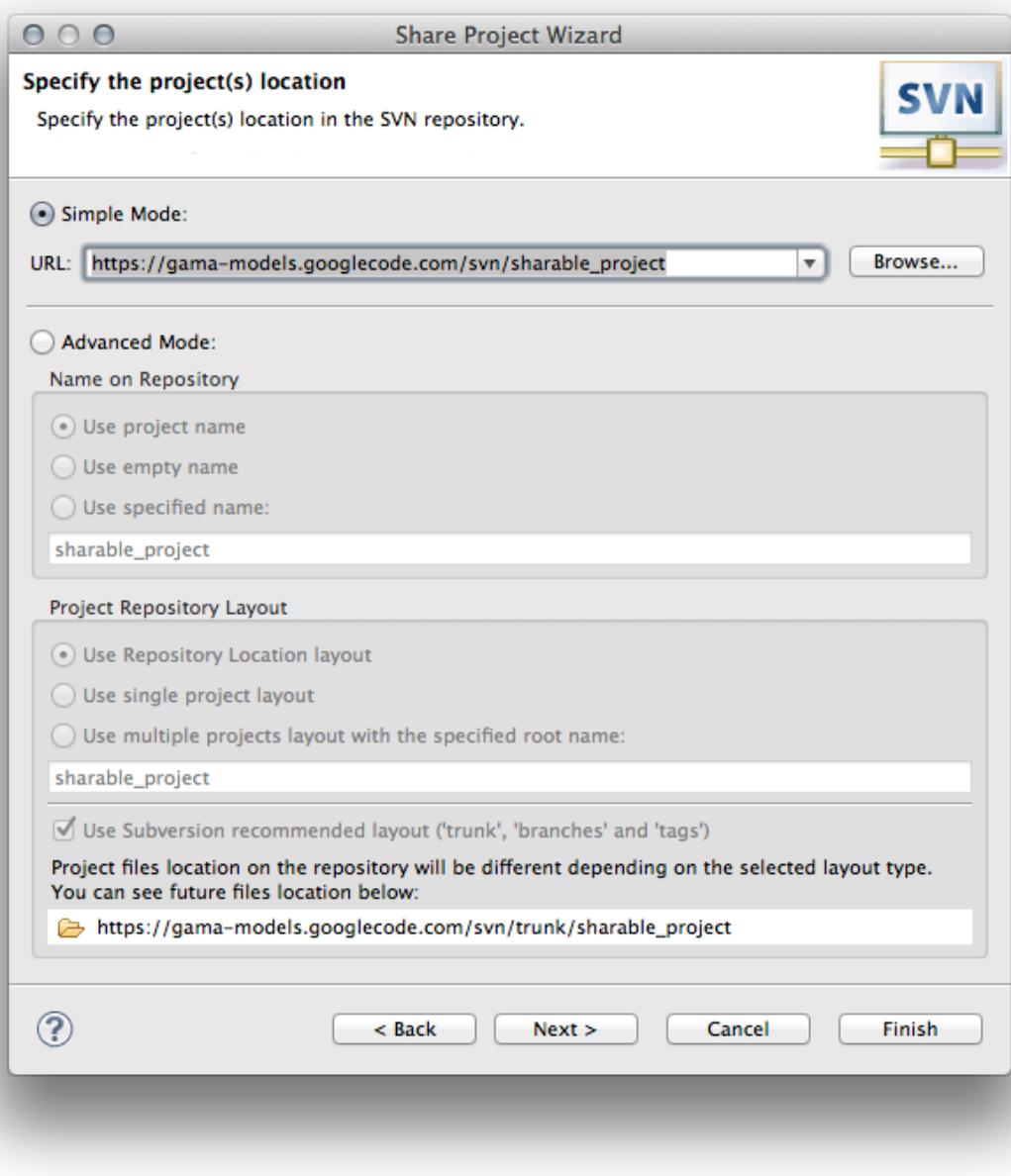
The initial sharing of contents between the GAMA platform and a remote repository can only be made at the level of a project. Once a project has been shared, however, its components are manageable on an individual basis (i.e. they can be moved, deleted, changed, and even removed from the sharing). All the commands related to sharing are grouped under a sub-menu ("Team") of the contextual menu of the [navigator](#). From there, it is possible to initiate sharing, stop it, but also manage the incoming and outgoing changes to a project. To initiate sharing, select the project you want to share and select "Team", then "Share Project...".



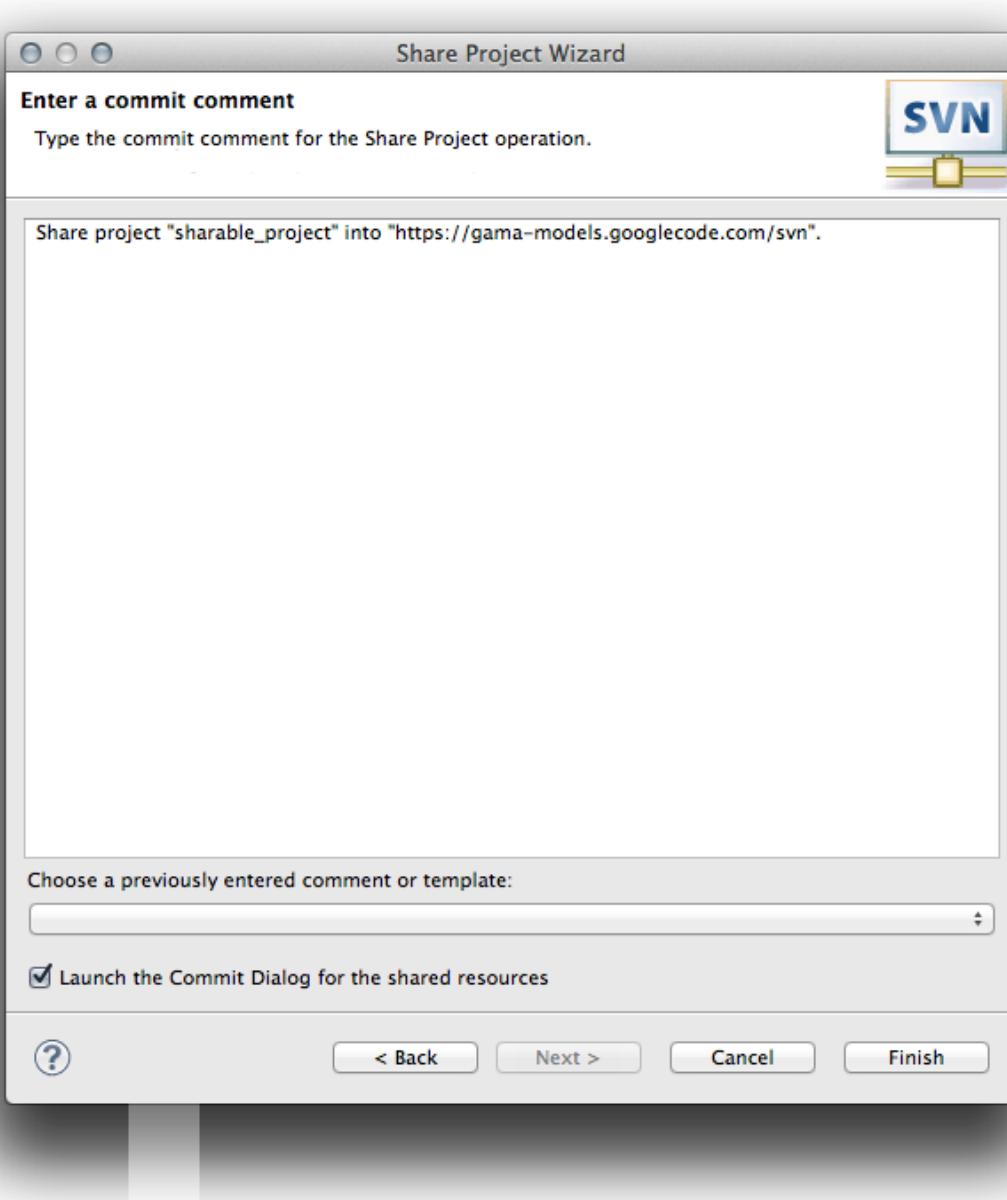
A dialog will then appear, asking you to enter a repository location. If you have already entered a repository location (in a previous sharing episode, for instance), you should be able to choose it from a list of existing locations. We suppose in this example that we enter the address and authentication information for the GAMA Models Repository (from which the "Shared Projects" category takes its information). If you plan to use another repository (with your co-workers, on a site like SourceForge or GoogleCode, etc.), these information will of course be different.



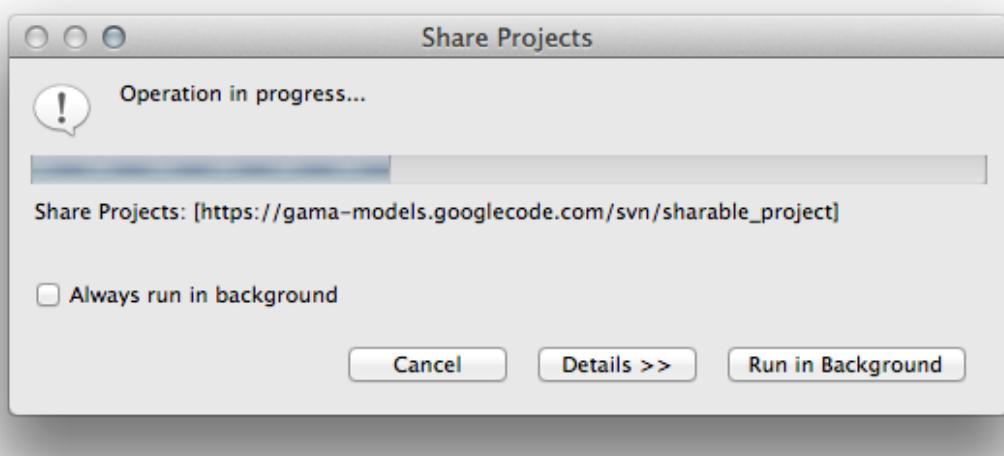
Once the repository location is validated, GAMA asks you for the name and structure to use for sharing the project on the repository. Always try to use "Simple Mode" if possible.



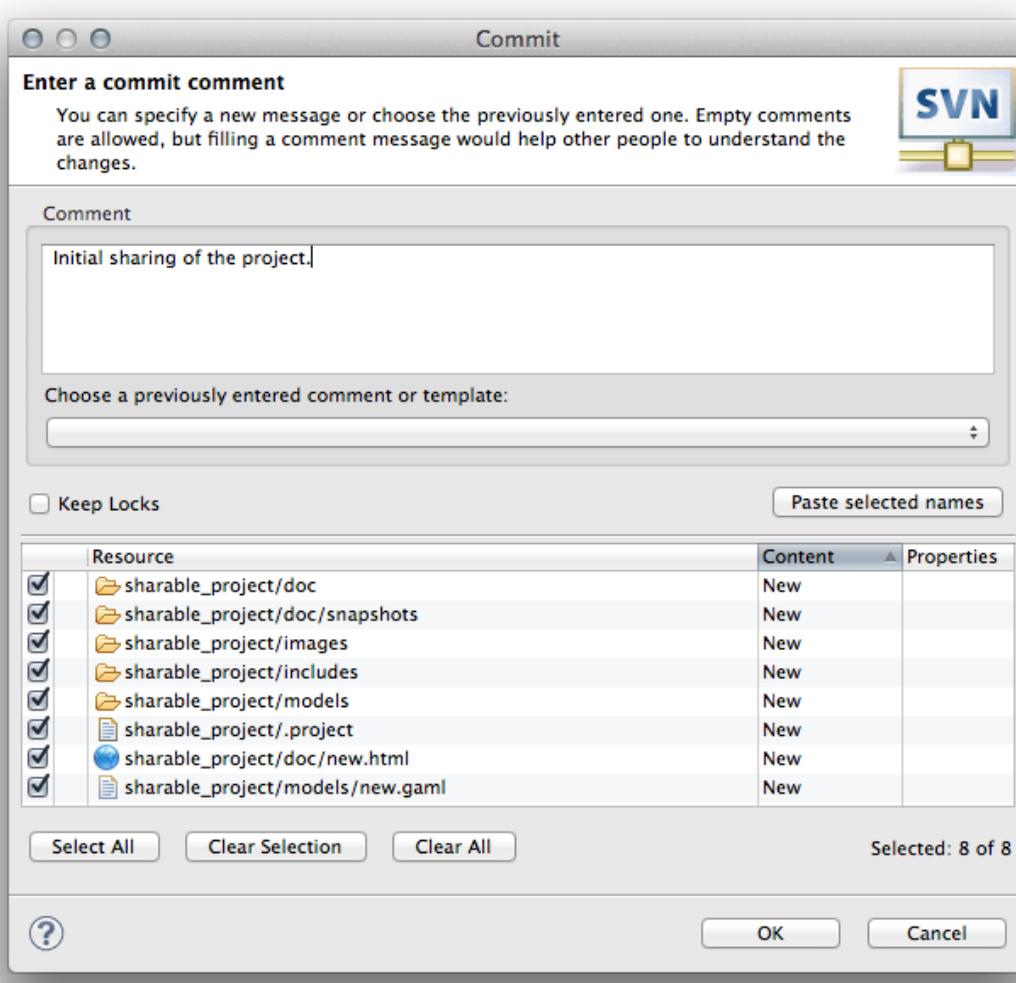
GAMA will then begin to *commit* the project (i.e. to create its initial structure in the repository). You can enter a comment at that stage, to easily identify the contents of this operation when someone will want to browse the changes made to the repository.



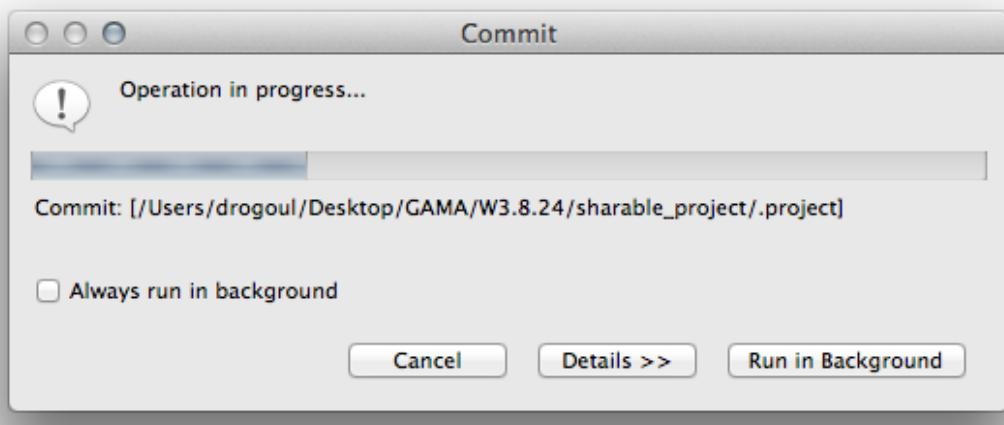
Creating the project structure on the repository takes a few seconds.



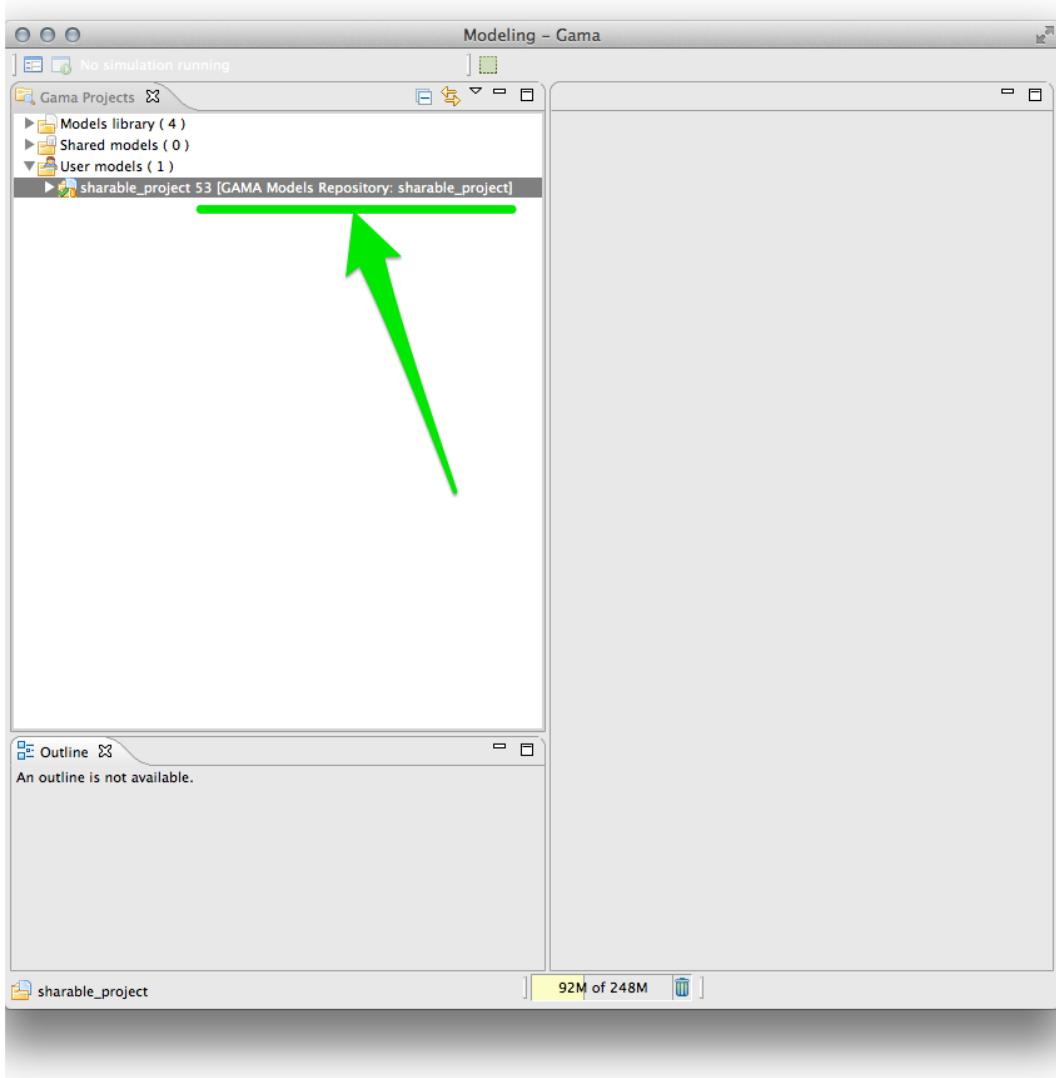
This phase is followed by a second, more important, *commit*, this time of the contents of the project. All the files present in the project will be copied to the repository. Once again, a comment can be entered at that stage.



GAMA displays a dialog while the files are individually copied to the repository.

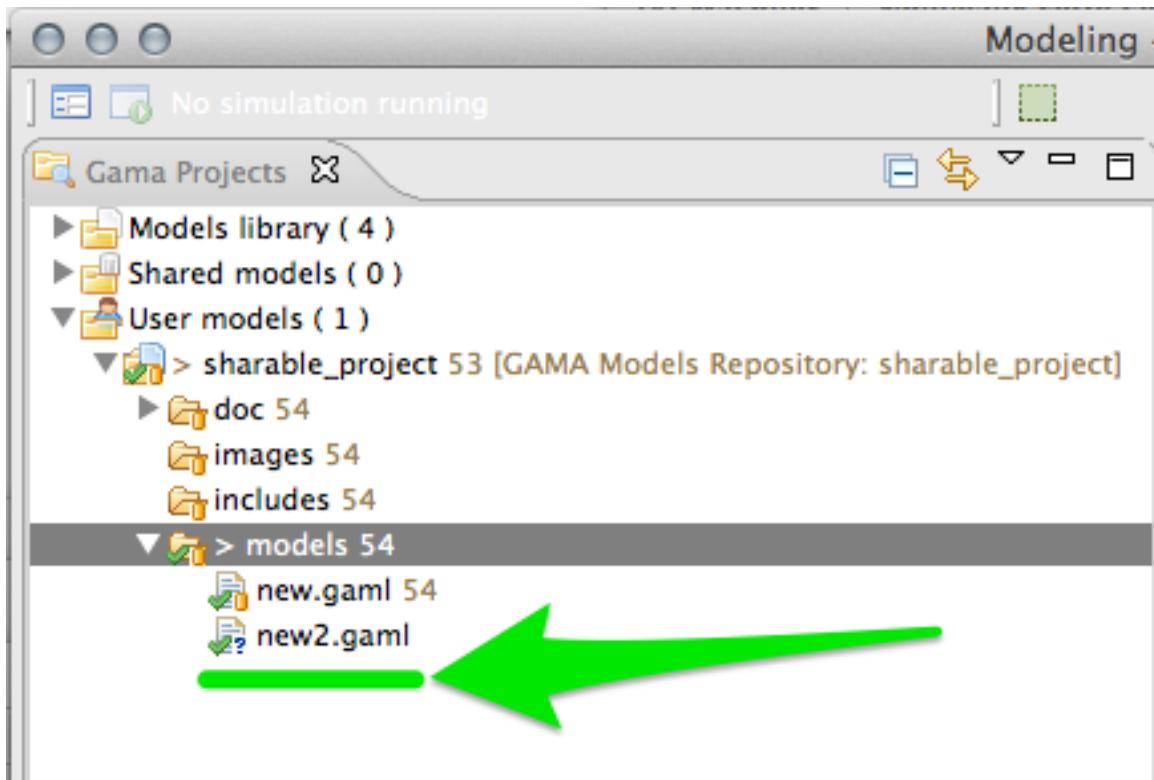


Once the initial sharing is over, the project sports new information in the navigator, namely its repository and *revision number* (i.e. a number beginning at 0 and incremented each time a change is made to the repository — not to the project).

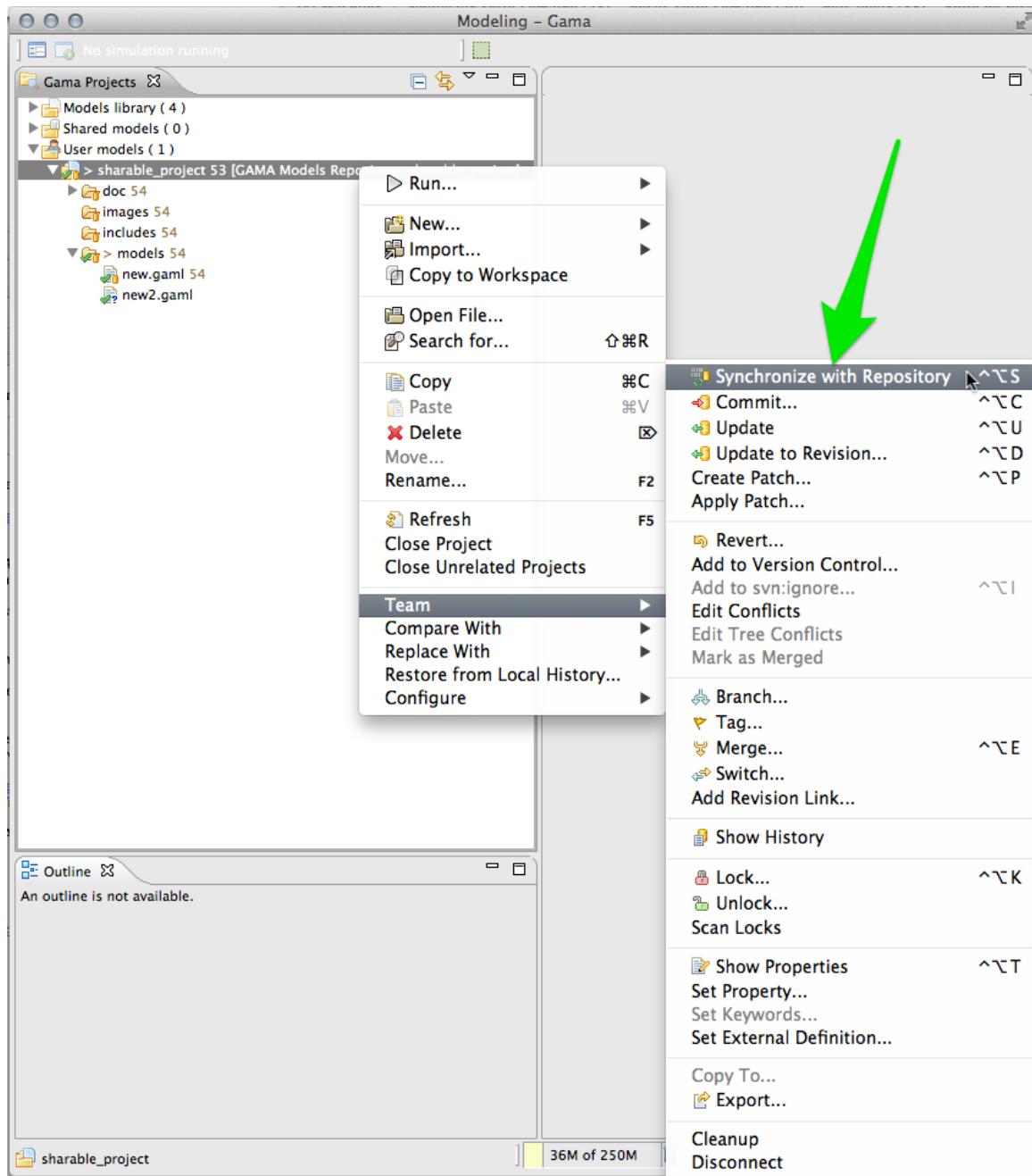


Committing Local Changes

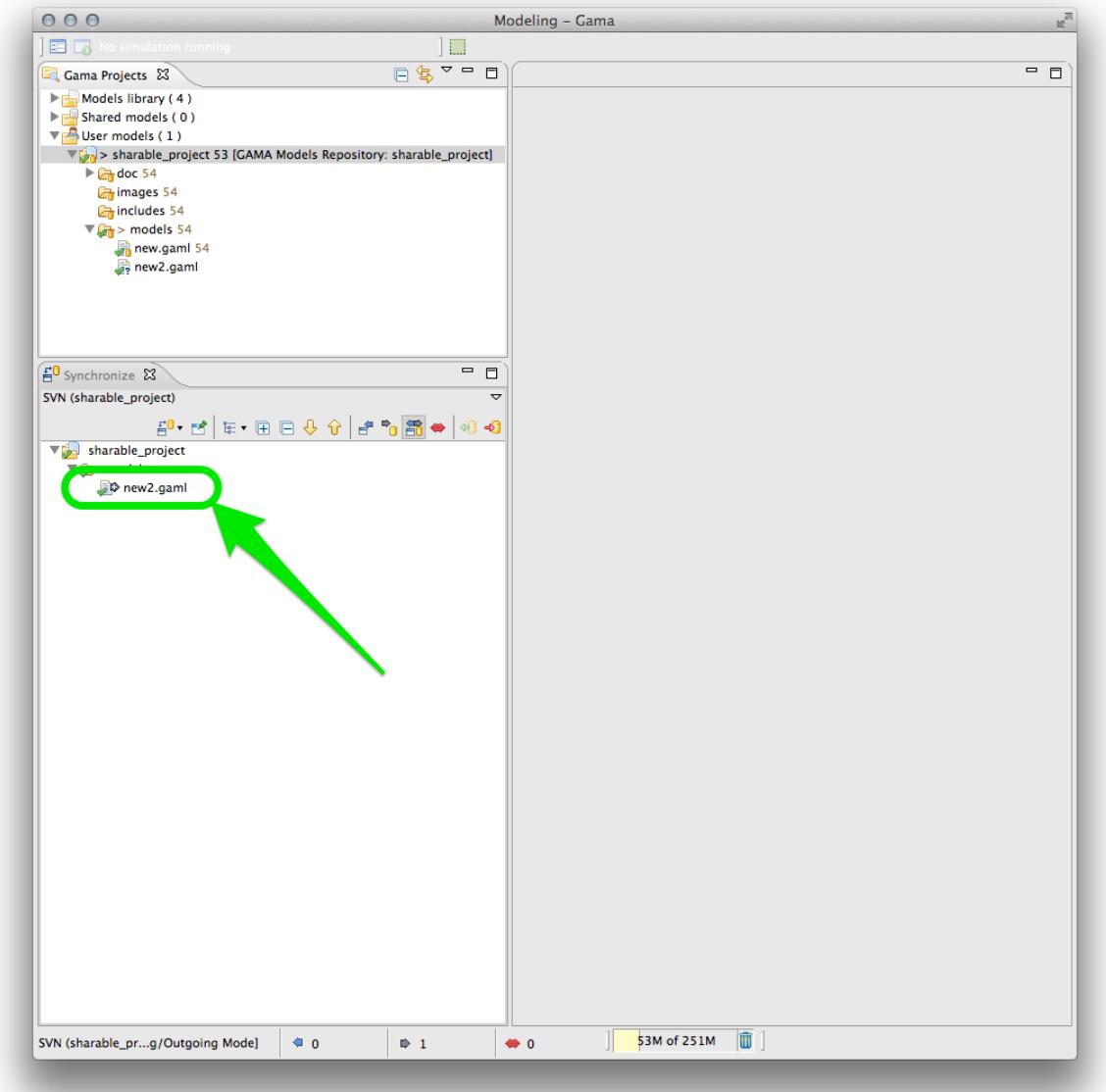
What happens when the shared project is locally modified, for example if we add a new model file to it ? You can see on the picture below that this file sports a different icon (with a question mark), indicating that it has no sharing information on its own (i.e. it does not exist in the repository but only locally). A modified, but already shared, file would have another different icon (with a star) to indicate that the local version is different from the one committed to the repository a while ago.



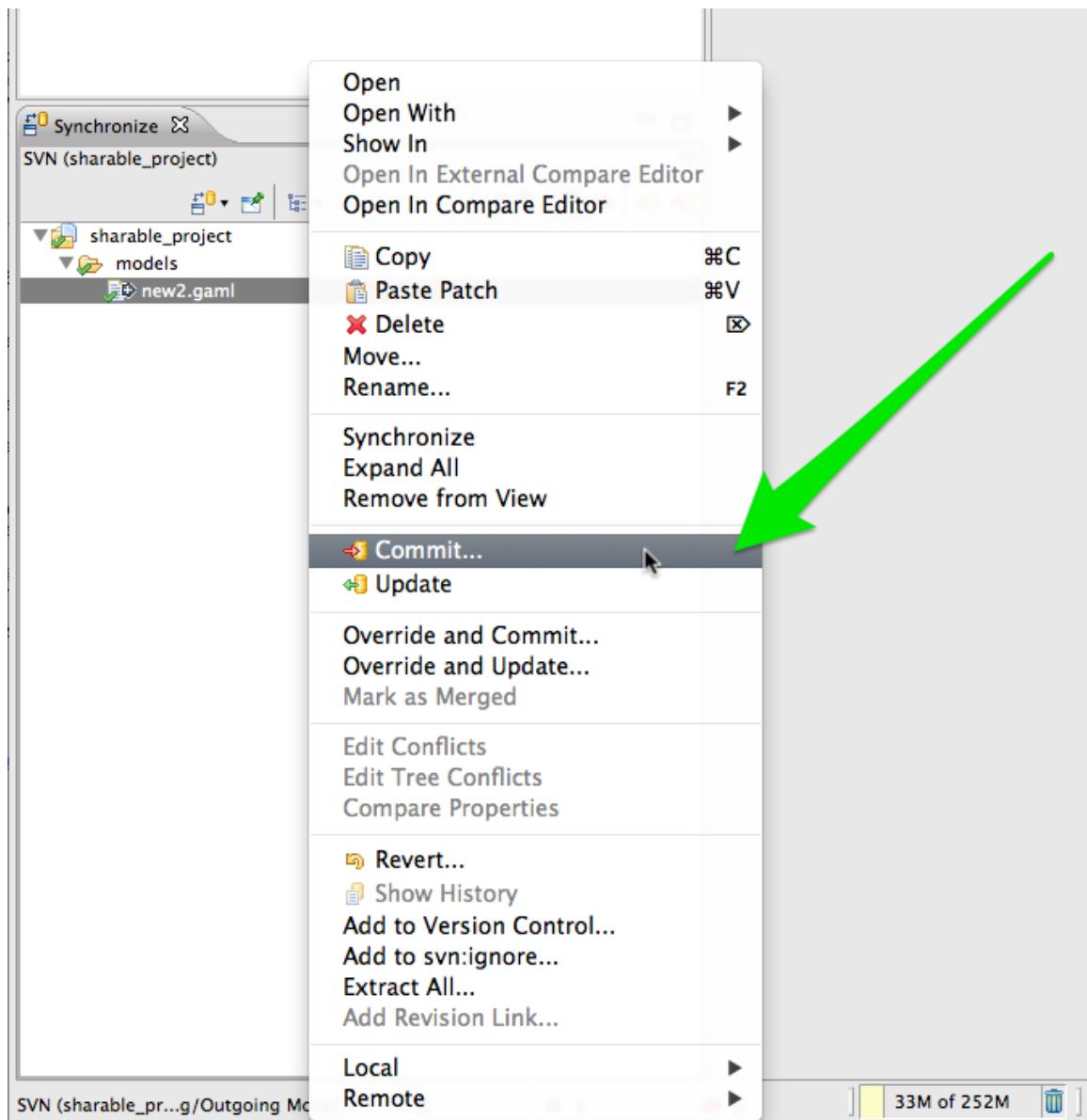
If these modifications constitute supplementary information that have to be shared as well, it can be done by using, again, the "Team" sub-menu (this time much more furnished, but explaining all the individual options is beyond the scope of this documentation. Please refer to the "official" documentations available on the top of this page). Here, we now invoke the "Synchronize with Repository" command. This command allows to browse the incoming (i.e. from the repository) and outgoing (i.e. from the platform) changes and it is always a good idea to use it instead of directly invoking "Commit.." or "Update...".



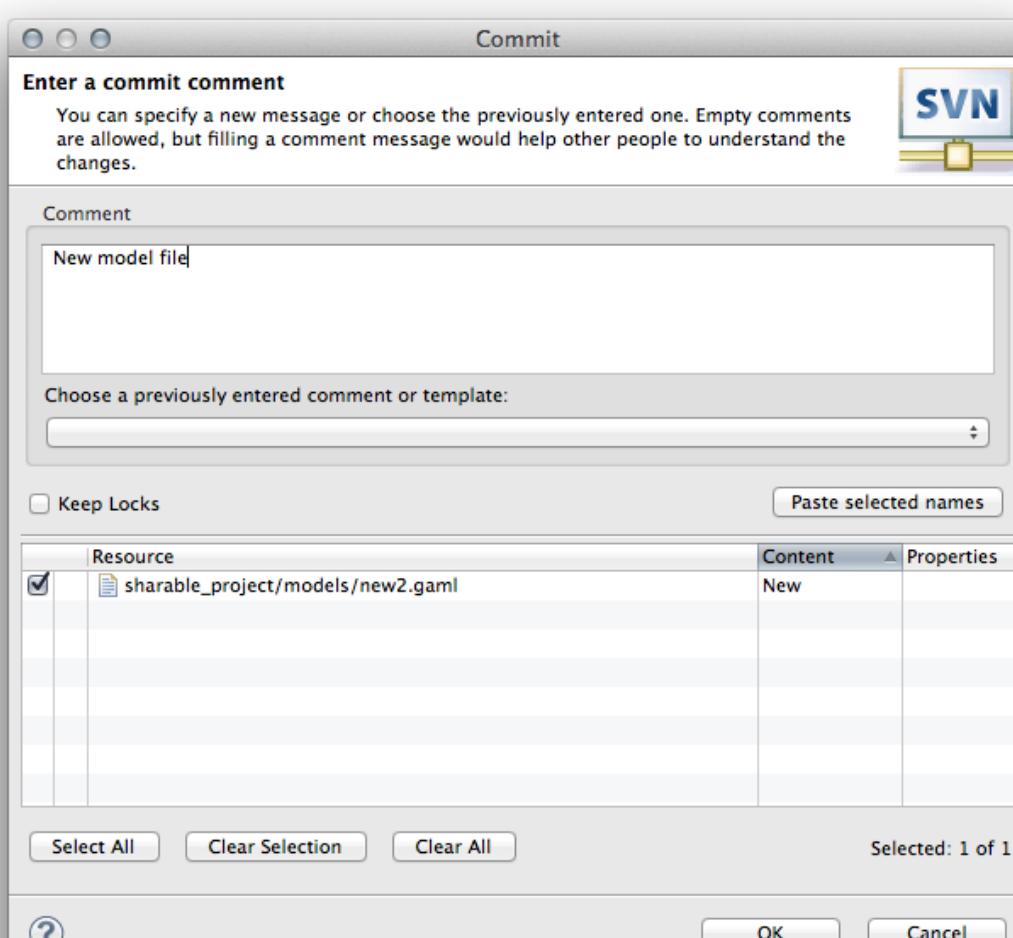
Selecting this command makes a new view (called "Synchronize") appear in the user interface. This view displays all the changes in both the repository and the platform, and indicates which files should be committed (sent to the repository) or updated (downloaded from the repository) in order for the two projects to be *synchronized*. Here, we see that the new model file is identified as a possible file to commit. In particular, GAMA does not identify any conflicts (which can happen when files are modified on both sides, and which must be solved manually).



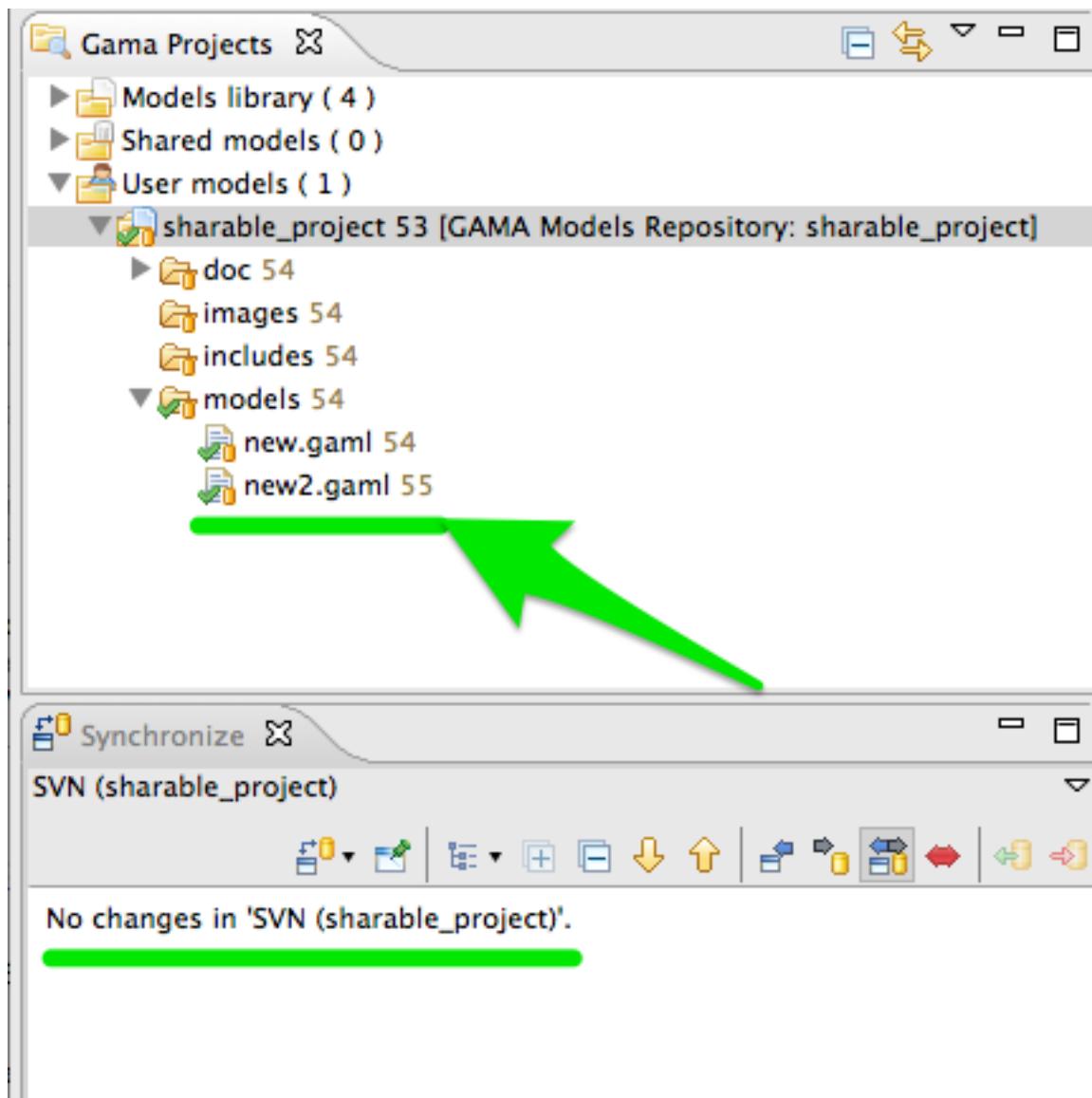
Selecting the new model file, we can then safely apply "Commit.." to it from its contextual menu.



GAMA opens a dialog listing all the outgoing changes and asks for a comment.



Once the commit is over, the new model file has disappeared from the "Synchronize view" (as it is now synchronized with its counterpart in the repository) and sports the same information as the others (with a difference: its *revision number* is logically higher).



Updating, i.e. applying incoming changes to the local copy of the projects or files, follows the same logic (except that "Update..." must be invoked in that case).

Sharing project within GAMA

GAMA developers can share directly their project within GAMA "shared models" element in the "Gama projects" tab. To do so one needs to upload its model to the gama-models svn (within the "trunk" directory : <https://gama-models.googlecode.com/svn/trunk>). A short description of the model is also required: create an **html** file within the *doc* directory named after your project name.

3. Editing Models

Introduction

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](#)
- 2. [Validation of Models](#)
- 3. [Graphical Editor](#)

3.1 GAML Editor

The GAML Editor

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.

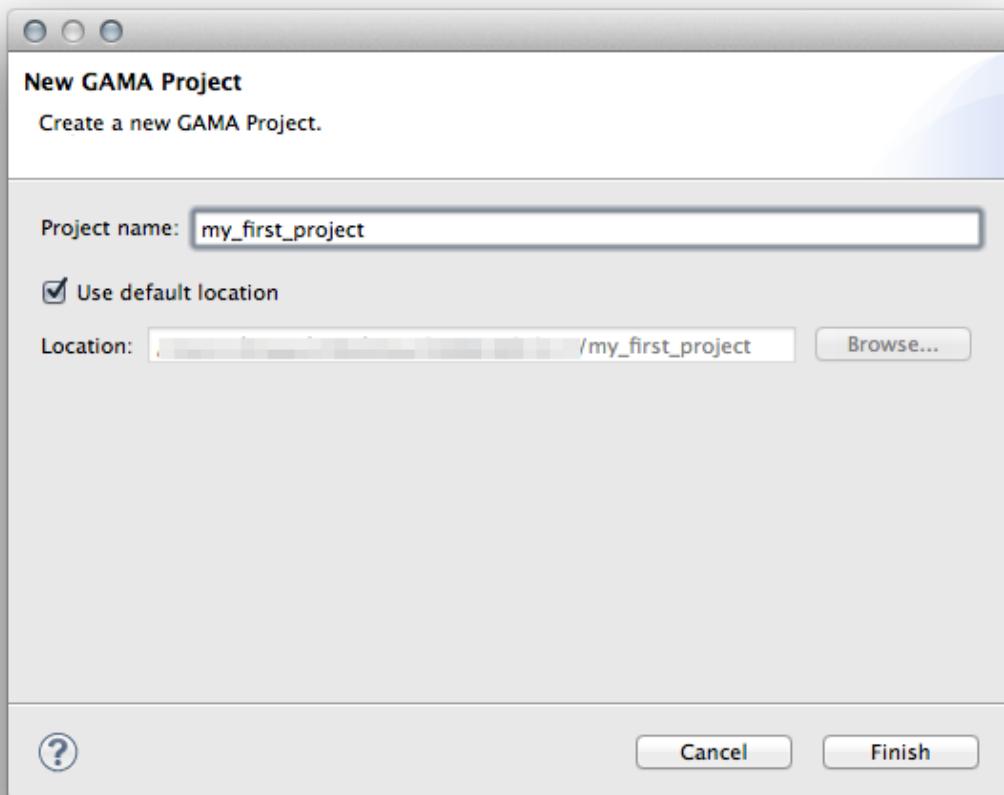
Creating a first model

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).

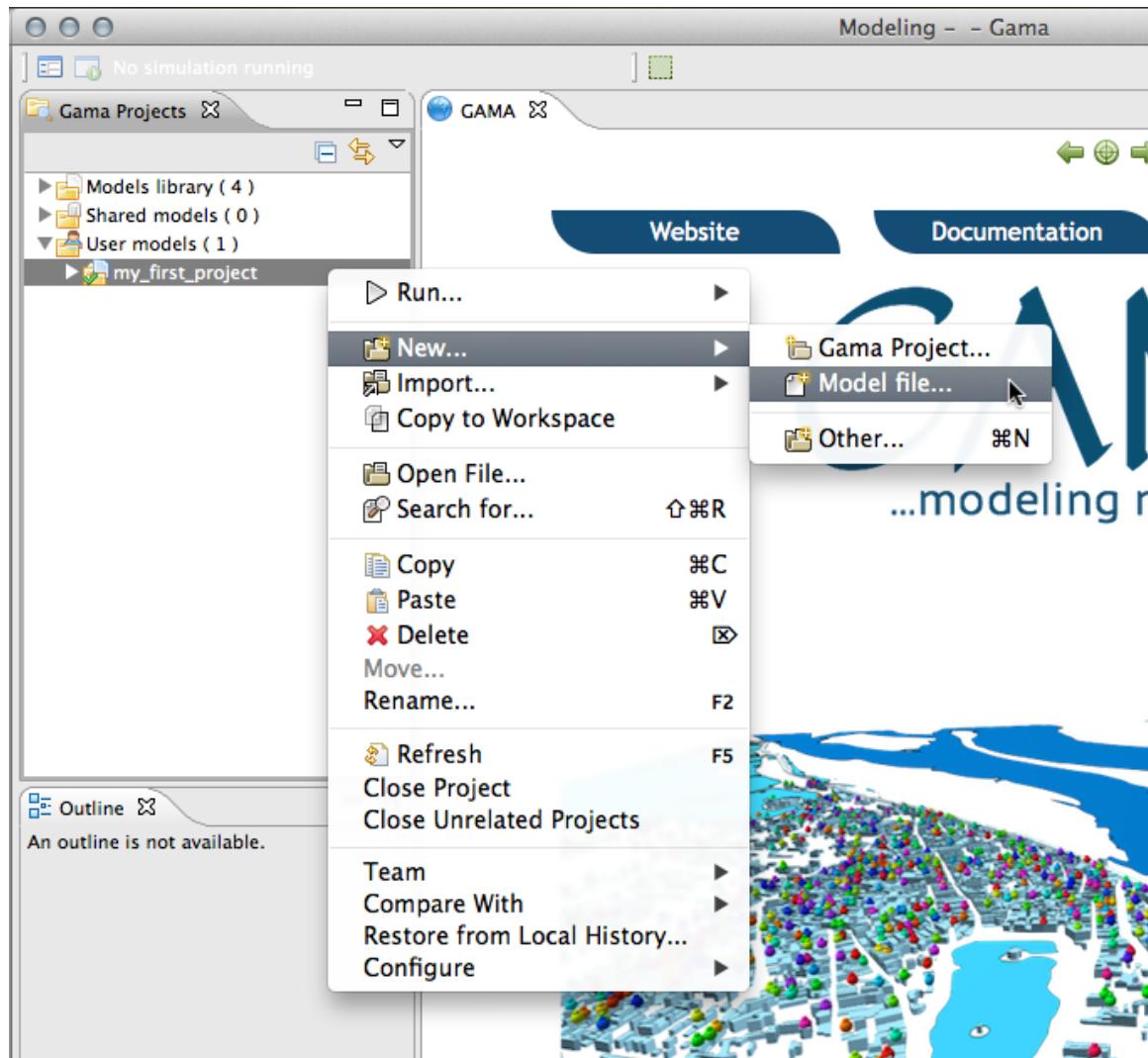


A dialog is then displayed, offering you to enter the name of the project as well as its location on the filesystem. Unless you are absolutely sure of what you are doing, keep the "Use default location" option checked. An error will be displayed if the project name already exists in the workspace,

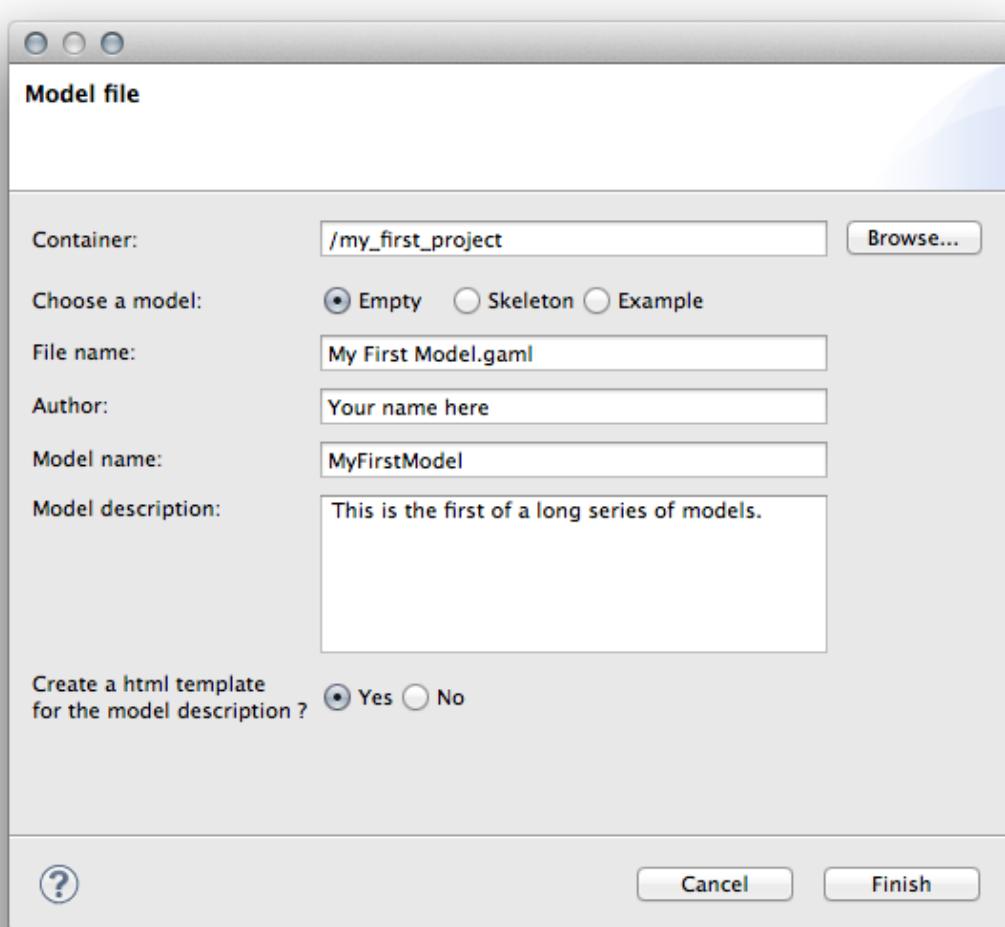
in which case you will have to change it. Two projects with similar names can not coexist in the workspace (even if they belong to different categories).



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.

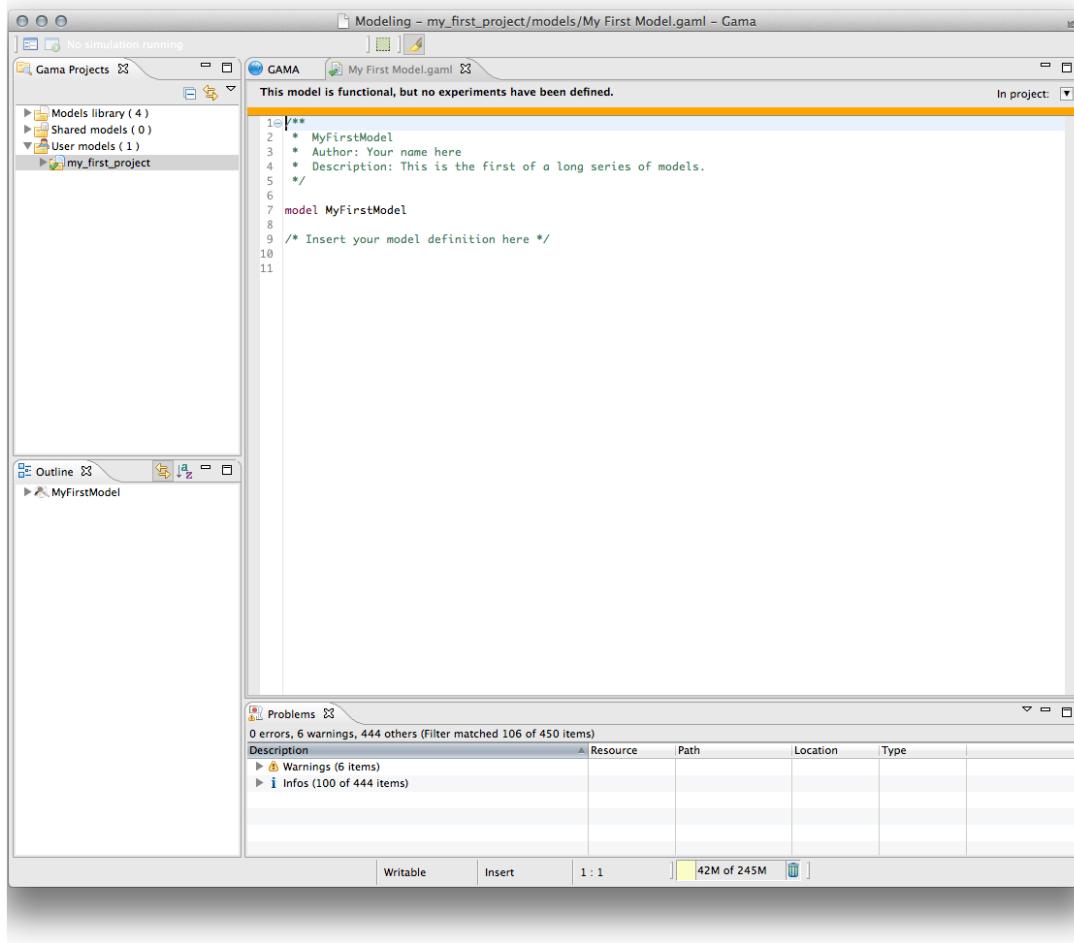


A new dialog is displayed, which asks for several required or optional information. The *Container* is normally the name of the project you have selected, but you can choose to place the file elsewhere. An error will be displayed if the container does not exist (yet) in the workspace. You can then choose 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.

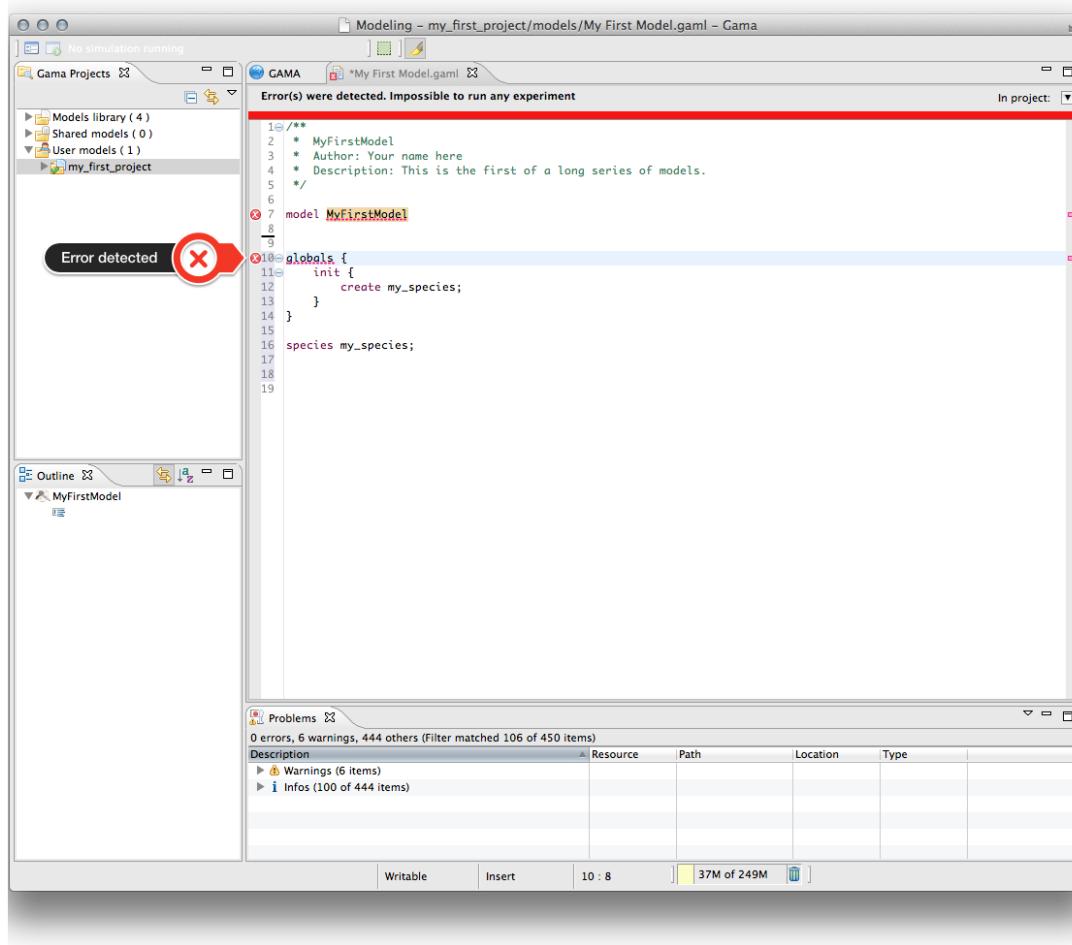


Status of models in editors

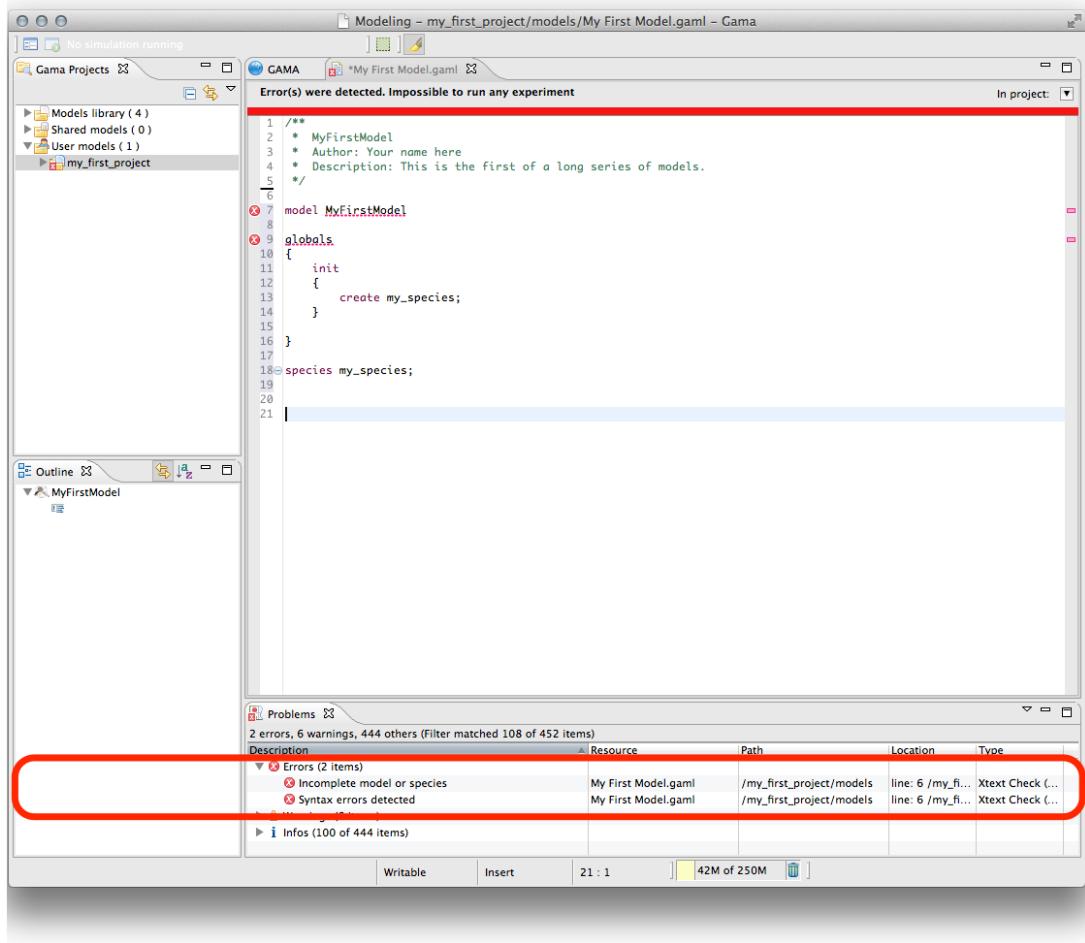
Once this dialog is filled and accepted, GAMA will display the new "empty" model.



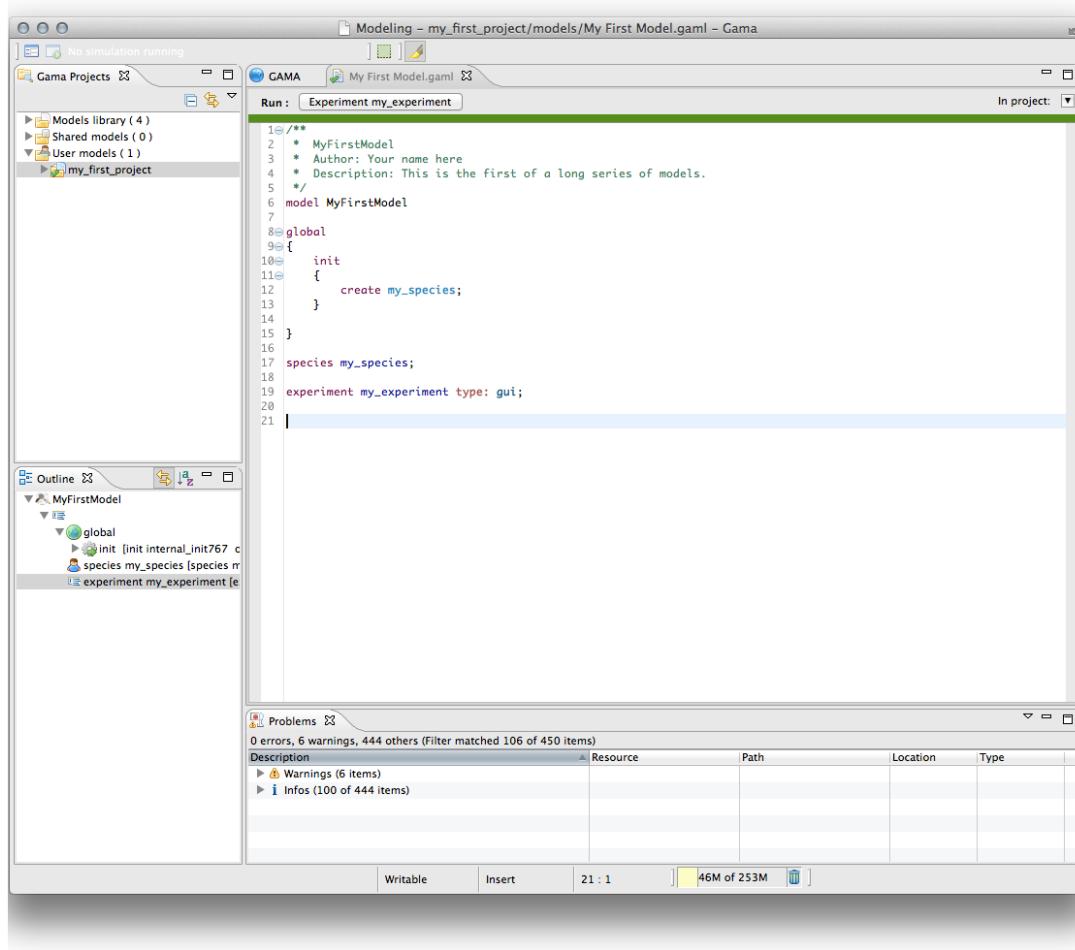
Although GAML files are just plain text files, and can therefore be produced or modified in any text processor, using the dedicated GAML editor offers a number of advantages, among which the live display of errors and model statuses. A model can actually be in four different states, which are visually 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).



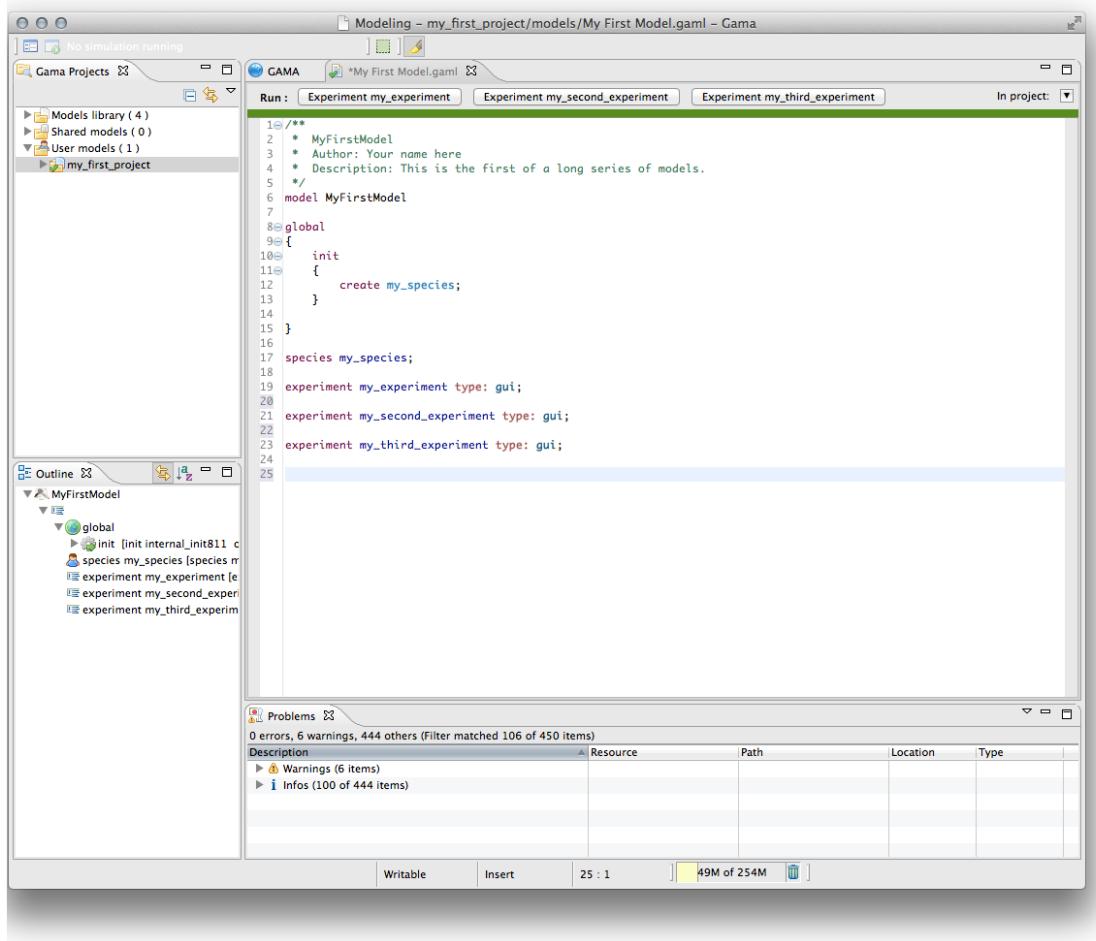
While the file is not saved, these errors remain displayed in the editor and nowhere else. If you save the file, they are now considered as "workspace errors" and get displayed in the "Problems" view below the editor.



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 launch this experiment on your model. See [the section about running experiments](#) for more information on this point.

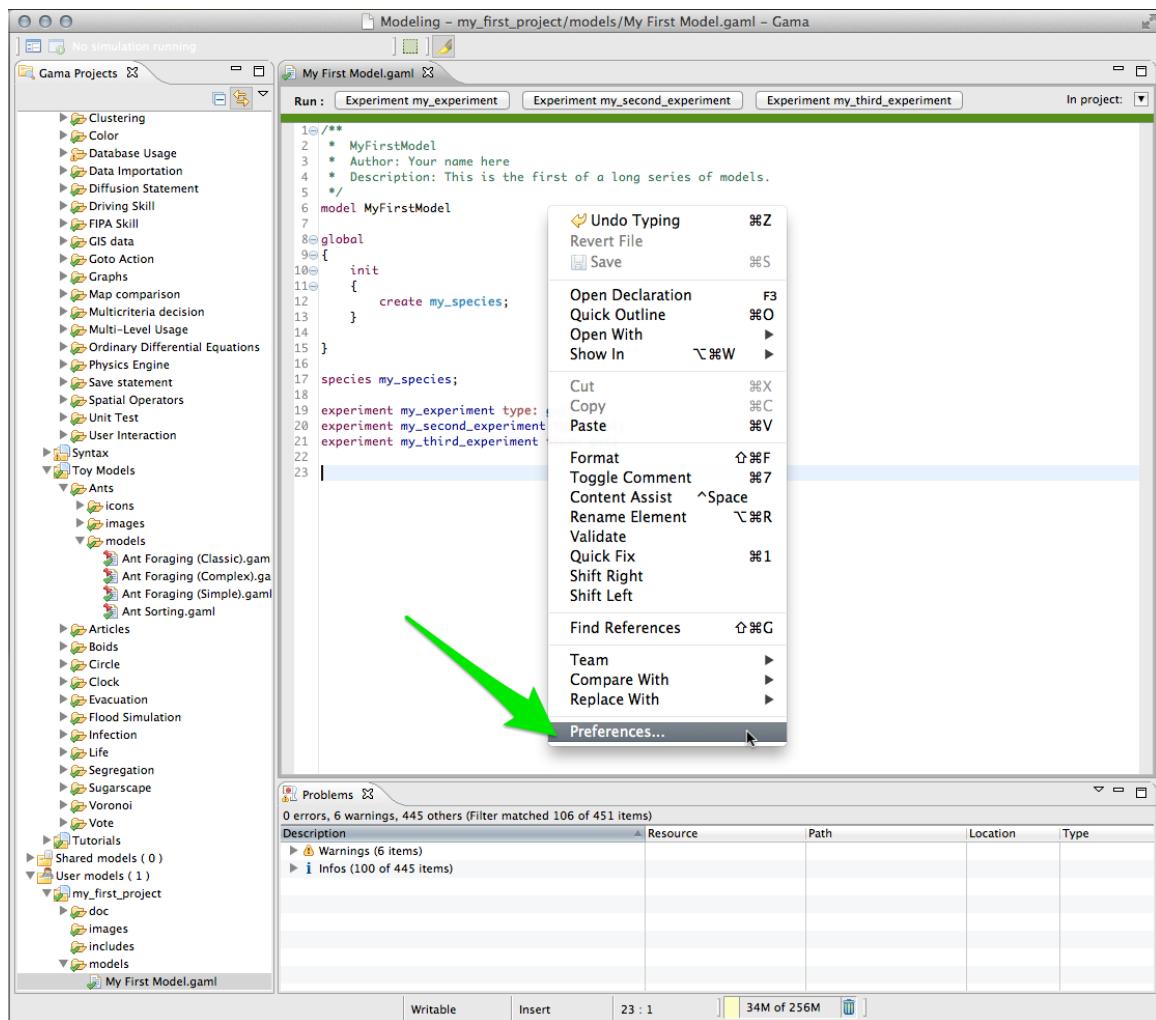


Experiment buttons are updated in real-time to reflect what's in your code. If more than one experiment is defined, corresponding buttons will be displayed in addition to the first one.

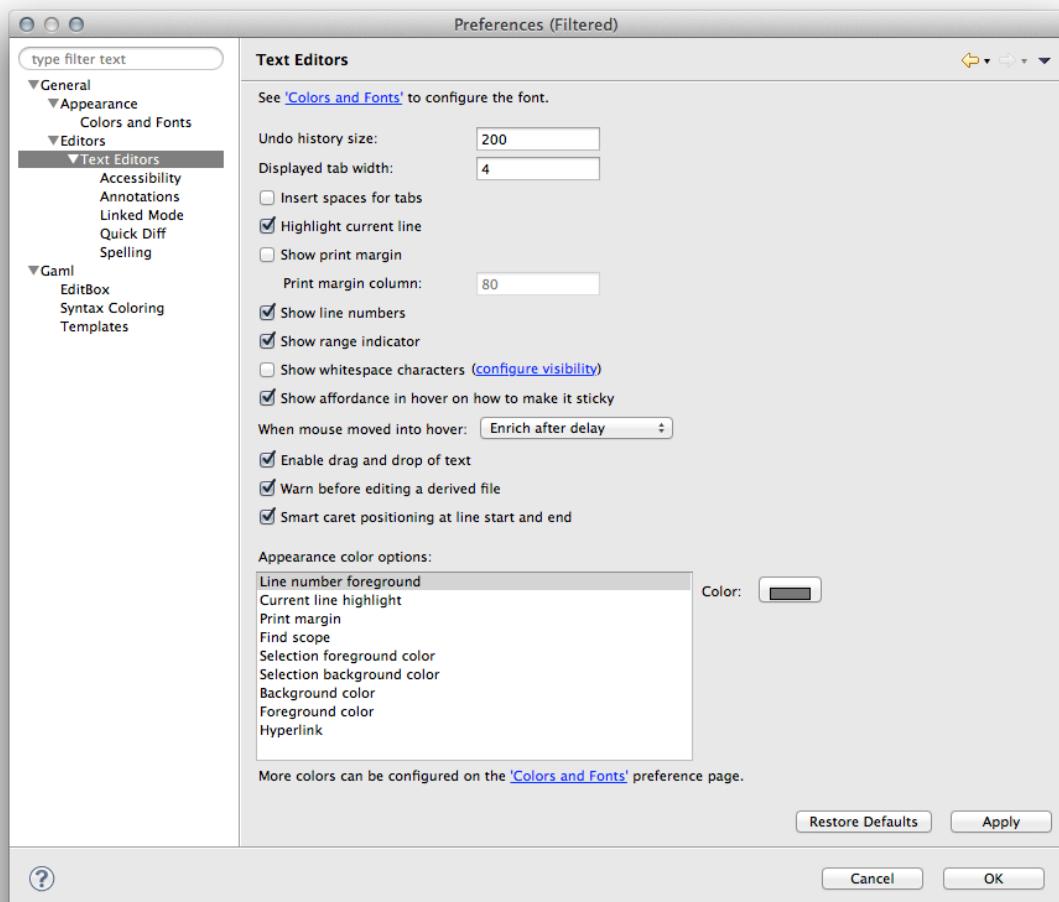


Editor Preferences

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.

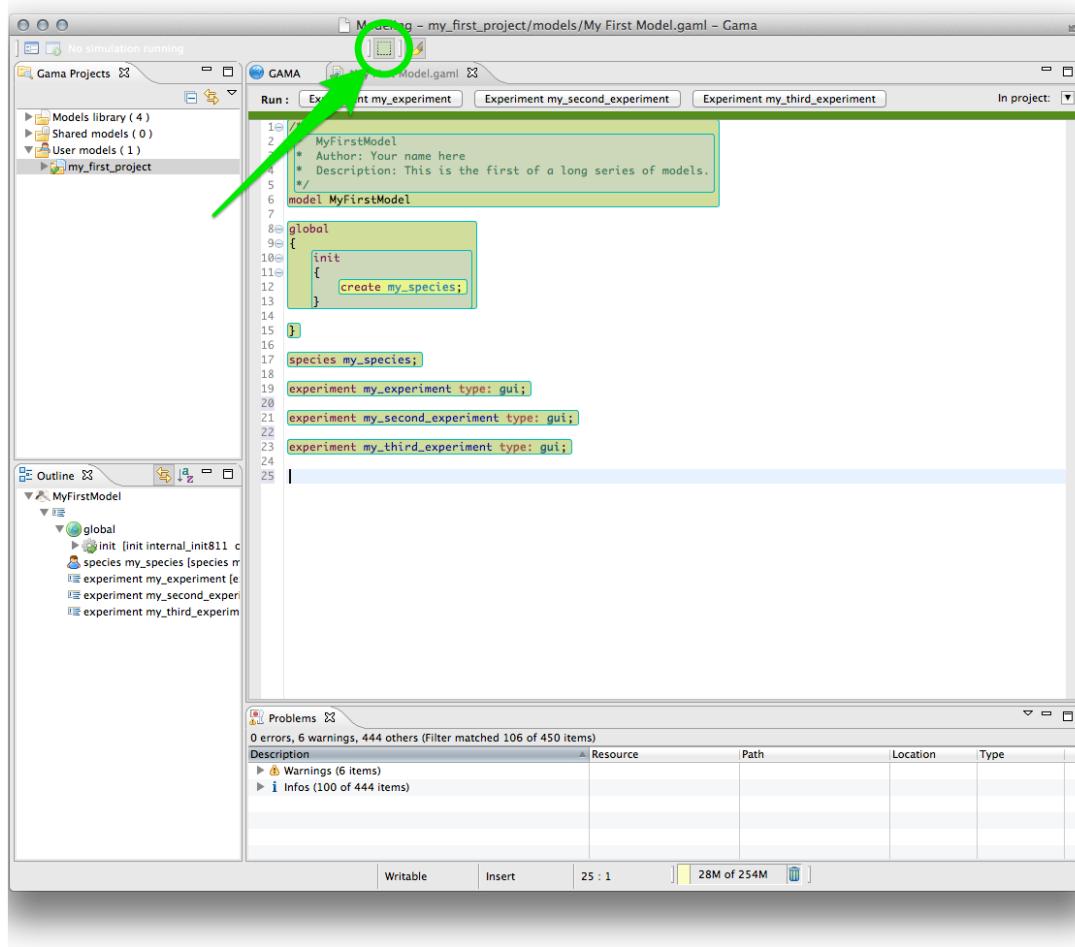


Explore the different items present there, keeping in mind that these preferences will apply to all the editors of GAMA and will be stored in your workspace.

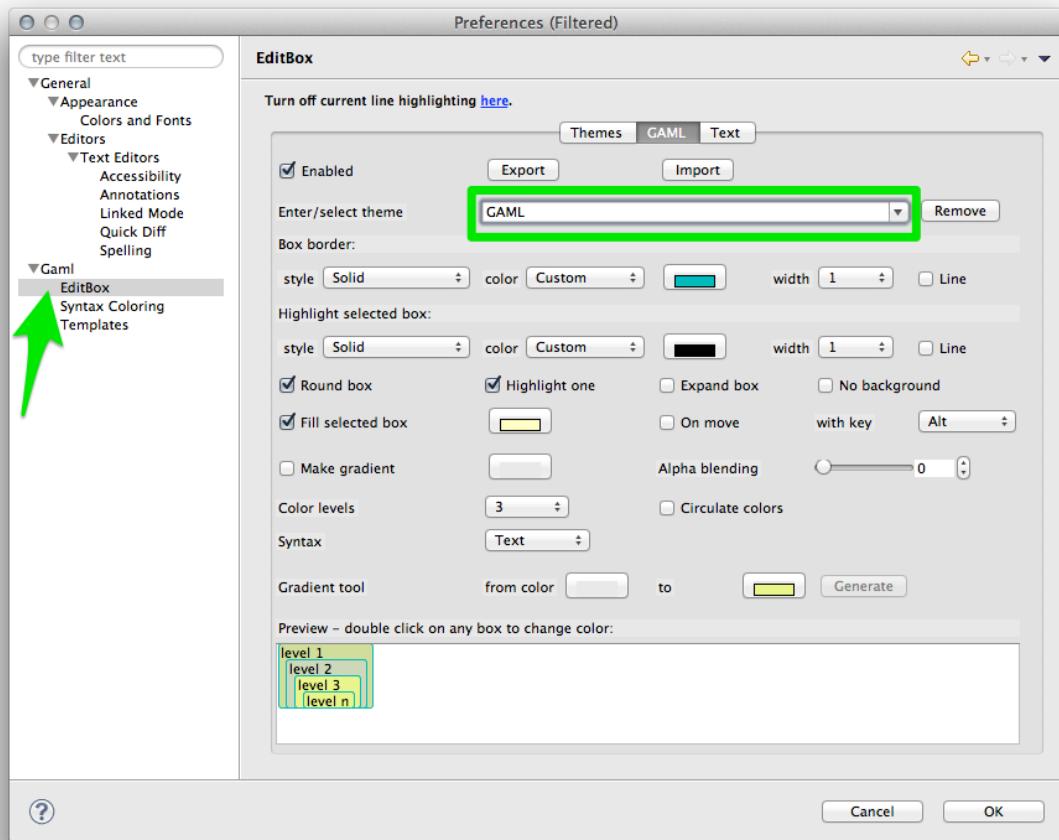


Structural highlighting

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.

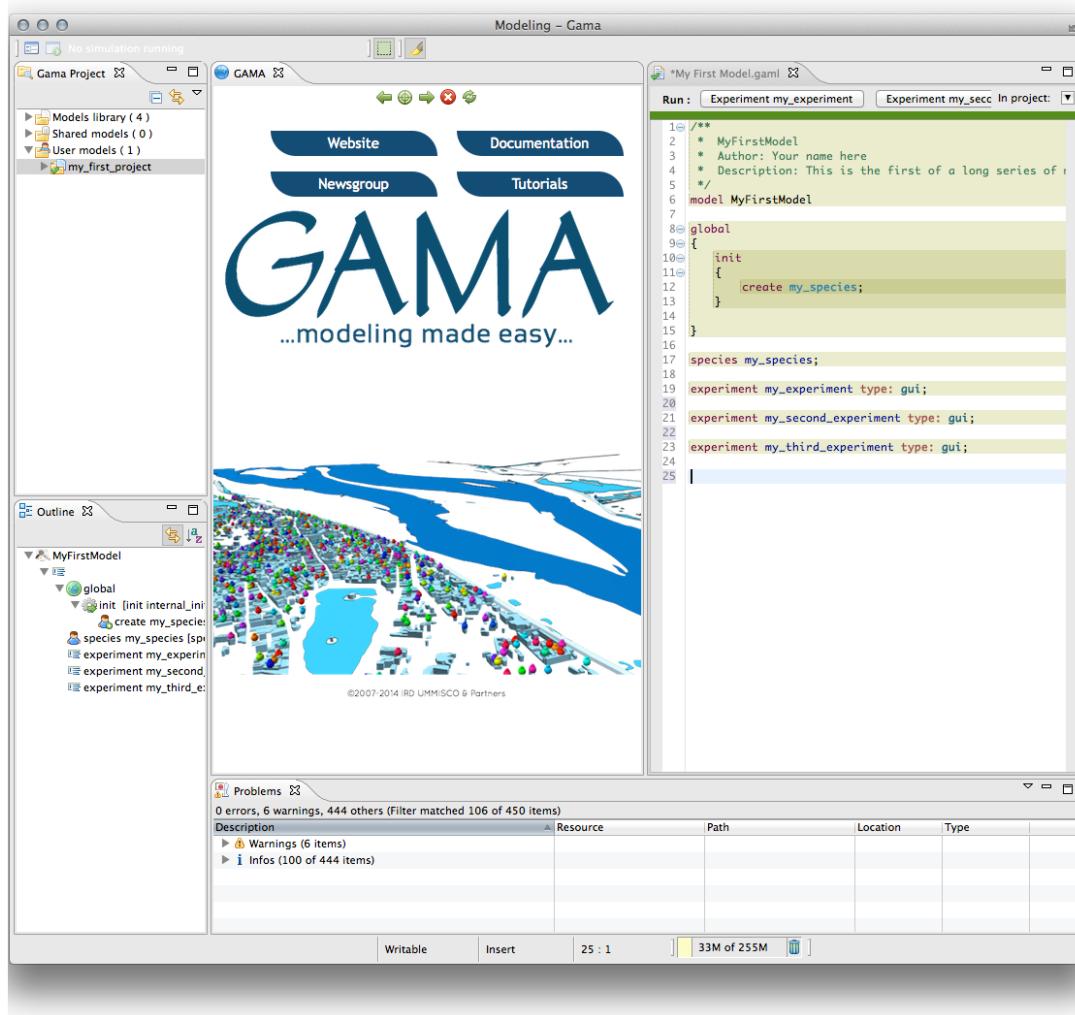


The Default theme of [EditBox](#) might not suit everyone's tastes, so the preferences allow to entirely customize how the "boxes" are displayed and how they can support the modeler in better understanding "where" it is in the code. The "themes" defined in this way are stored in the workspace, but can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



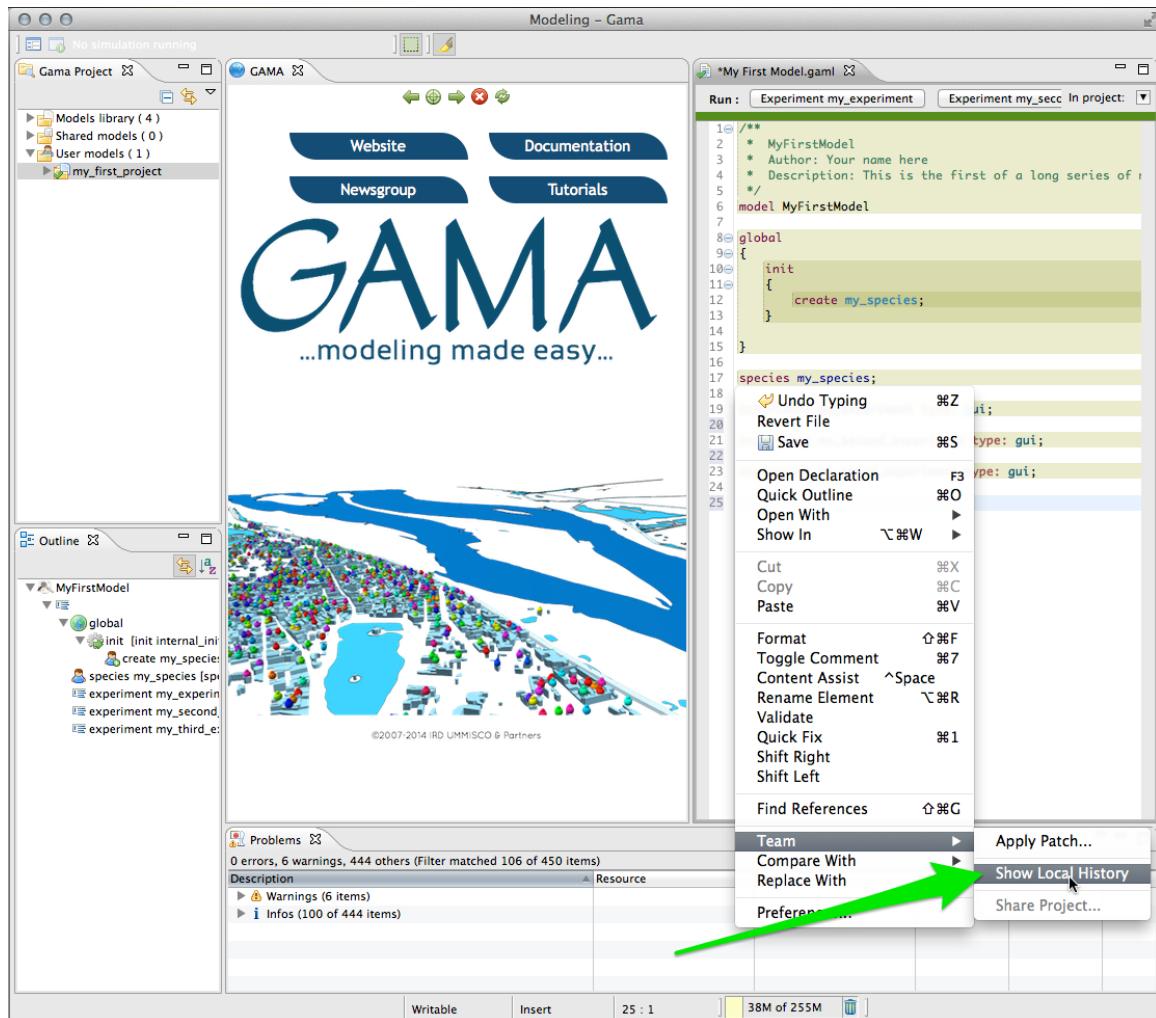
Multiple editors

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 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.

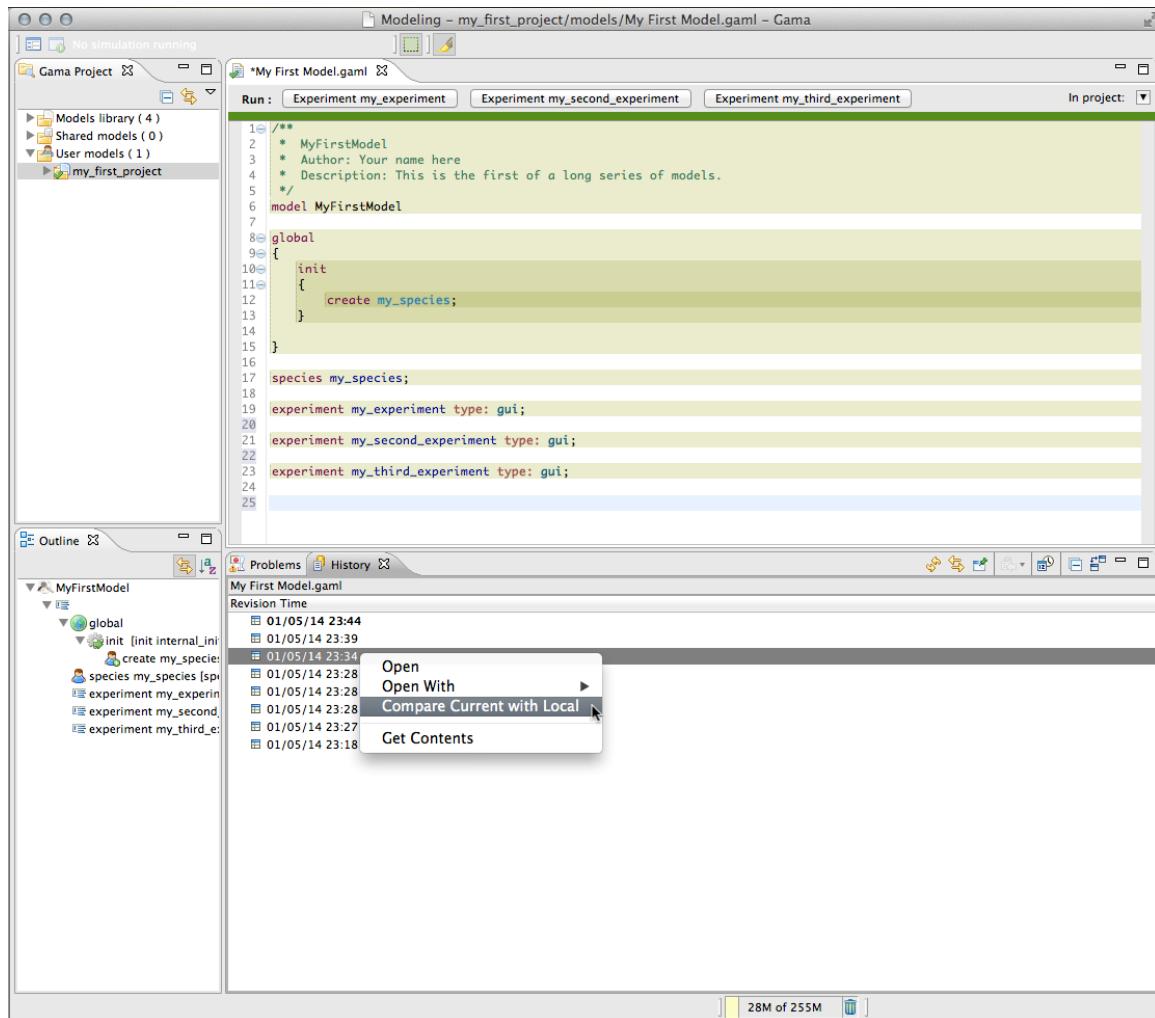


Local history

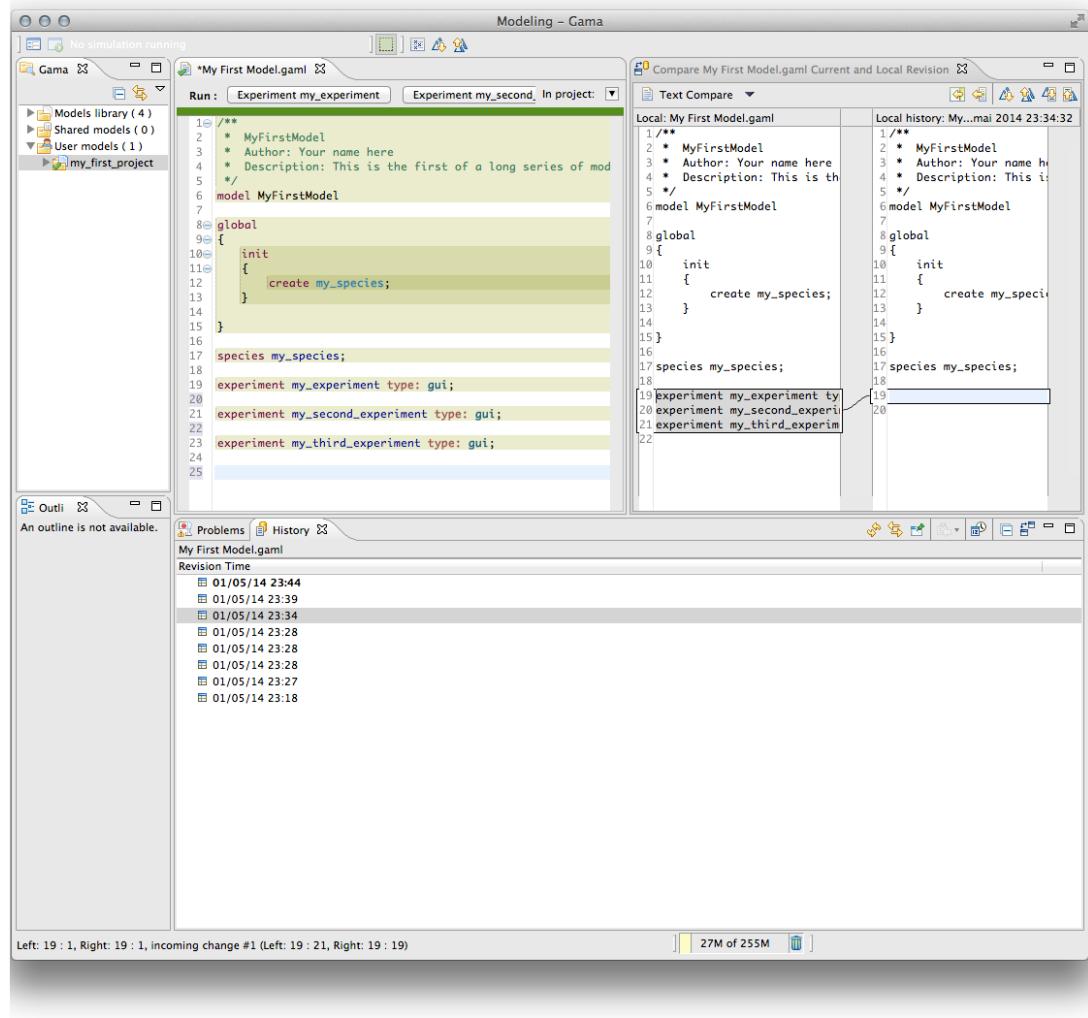
Among the various options present to work with models, which you are invited to try out and test at will, one, called *Local history* is particularly interesting and worth a small explanation. When you edit models, GAMA keeps in the background all the successive versions you save (the history duration 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 !



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.

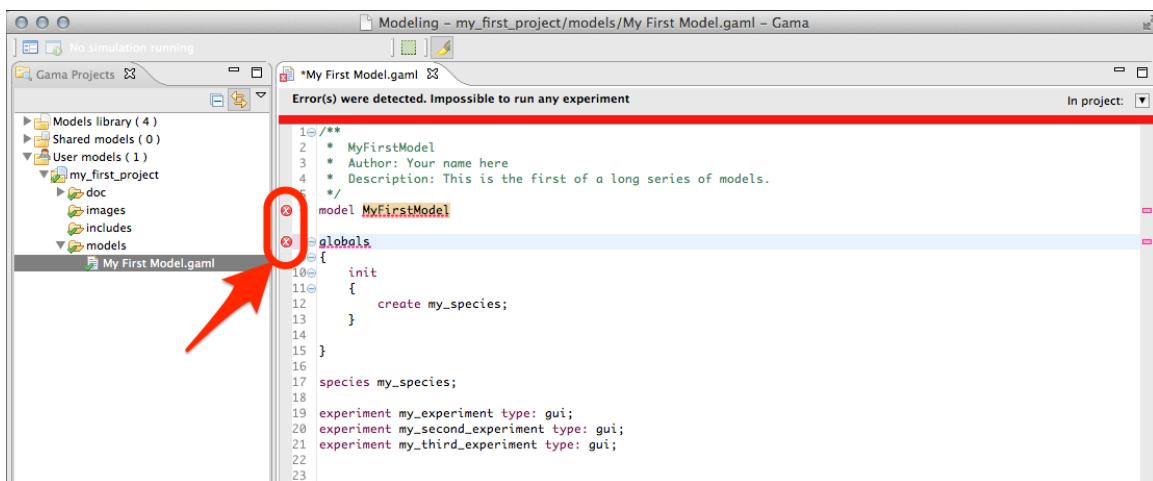
3.2 Validation of Models

Validation of Models

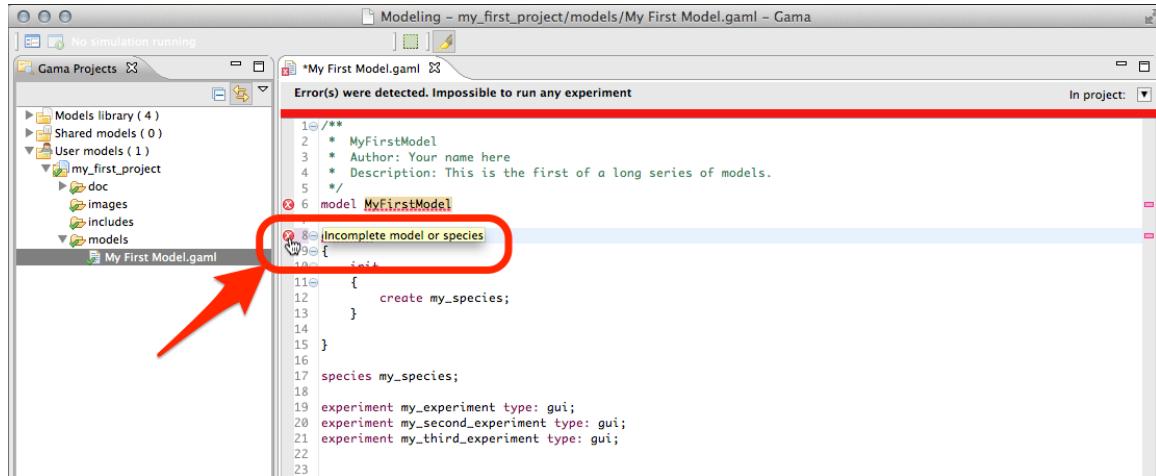
When editing a model, GAMA will continuously validate (i.e. *compile*) what the modeler is entering and indicate, with specific visual affordances, various information on the state of the model. This information ranges from documentation items to errors indications. We will review some of them in this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

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.



Hoving 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

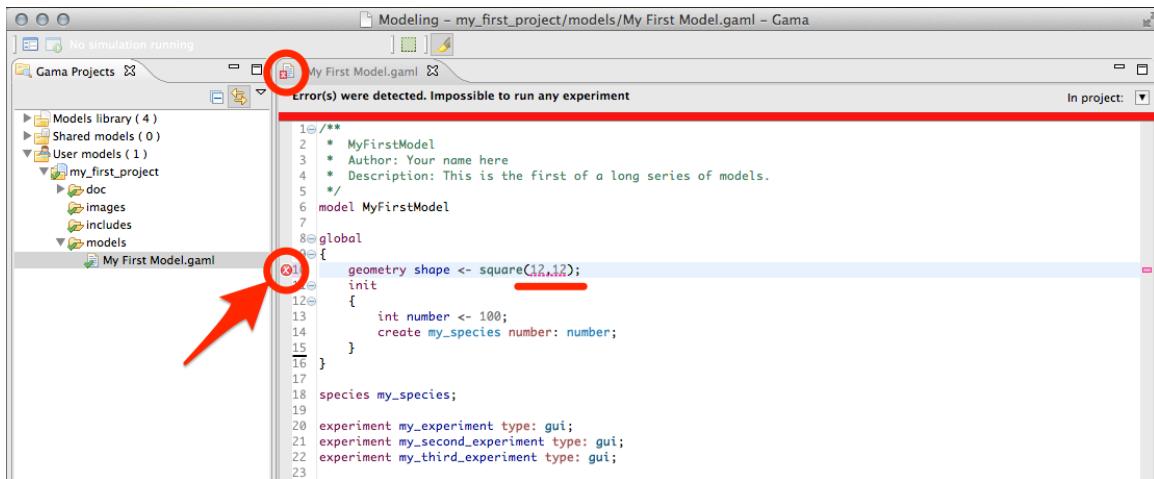
When syntactic errors are eliminated, the validation enters a so-called semantic phase, during which it ensures that what the modeler has written makes sense with respect to the various rules of the language. To understand the difference between the two phases, take a look at the following example. This sentence below is **syntactically** correct:

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

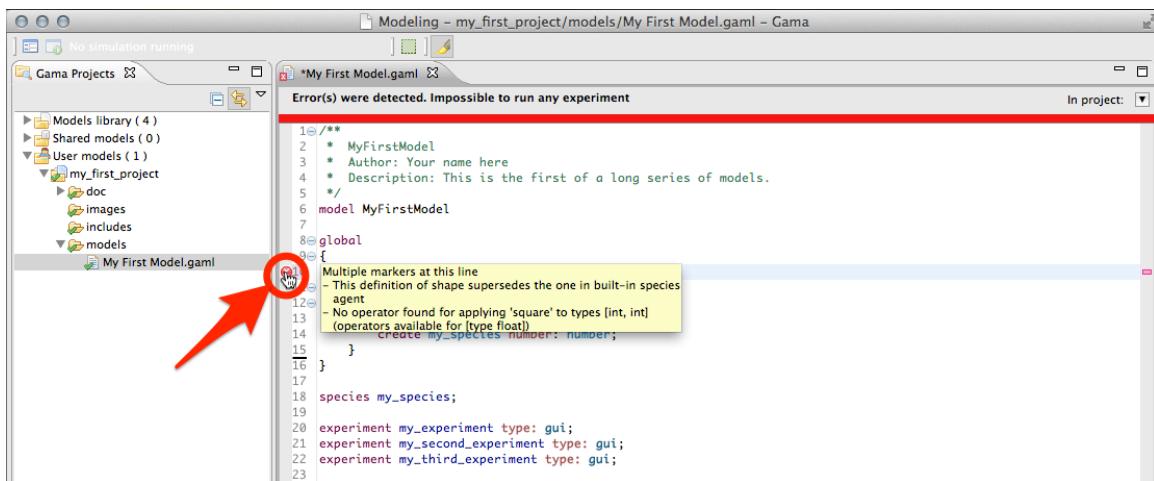
But it is **semantically** incorrect because a species cannot be parent of itself. No syntactic errors will be reported here, but the validation will fail with a **semantic** error.



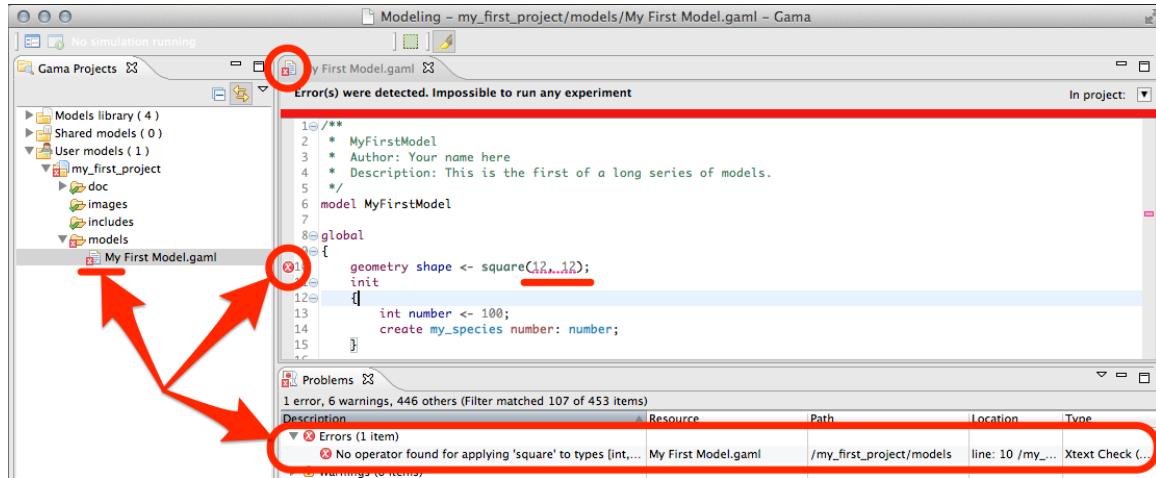
Semantic errors are reported in a way similar to syntactic errors, except that no **marker** are displayed beside the model statement. The compiler tries to report them as precisely as possible, underlining the places where they have been found and outputting hopefully meaningful error messages. In the 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 experimental.



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).

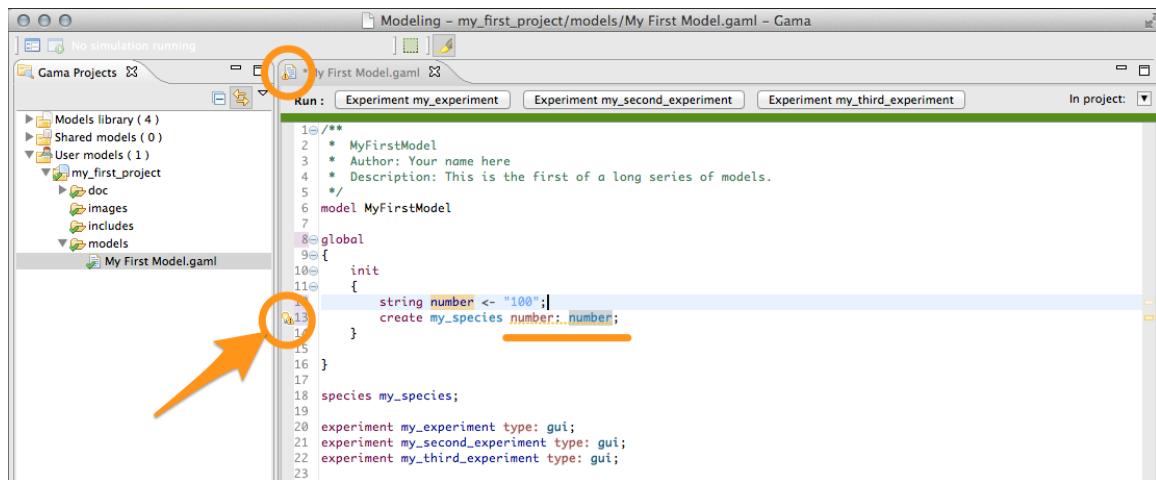


While the editor is in a so-called *dirty* state (i.e. the model has not been saved), errors are only reported locally (in the editor itself). However, as soon as the user saves a model containing syntactic or semantic errors, they are "promoted" to become workspace errors, and, as such, indicated in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

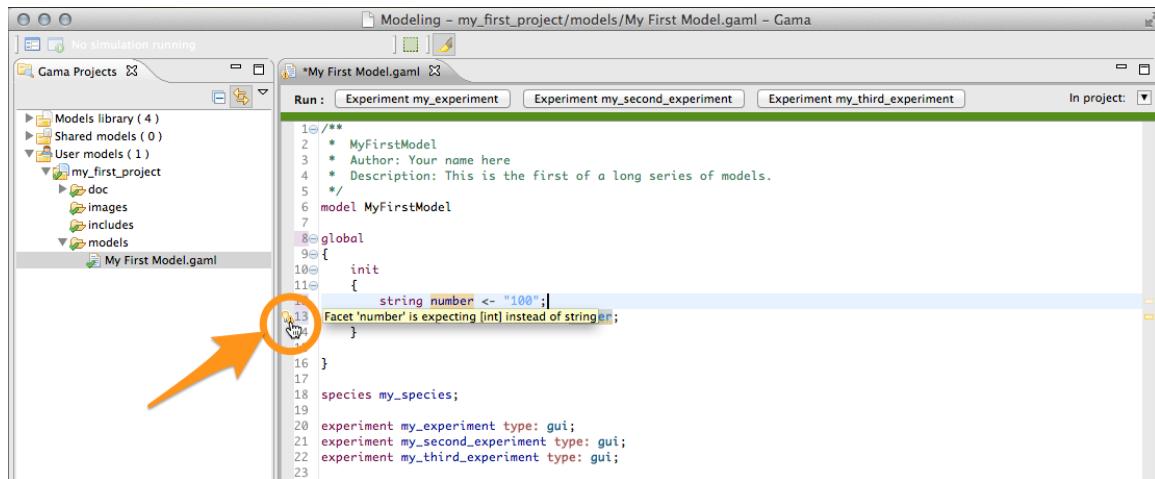


Semantic warnings

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 some flaws in the logic of the model.

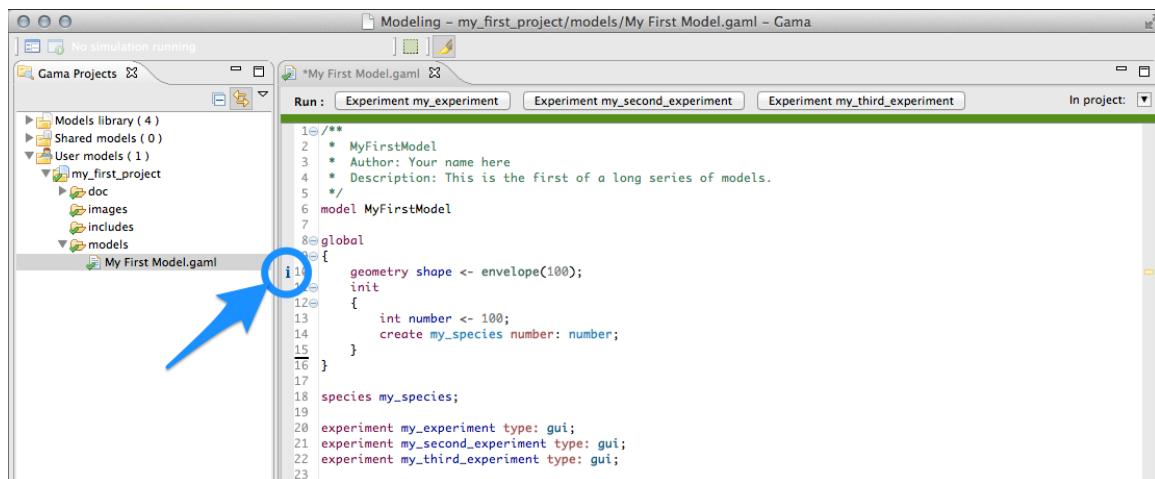


Hoving over the warning **marker** will allow the modeler to have access to the explanation and hopefully fix the cause of the warning.

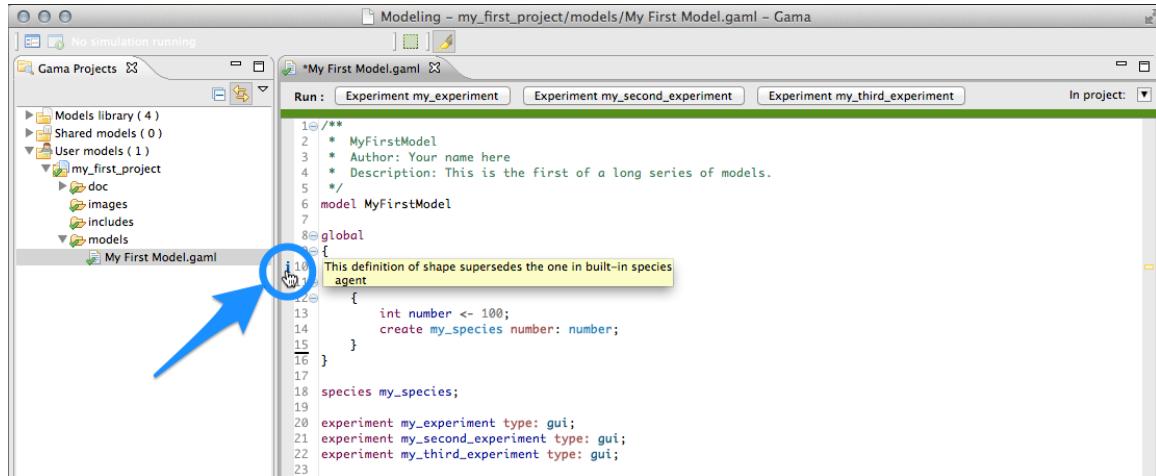


Semantic information

Besides warnings, another type of harmless feedback is produced 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).

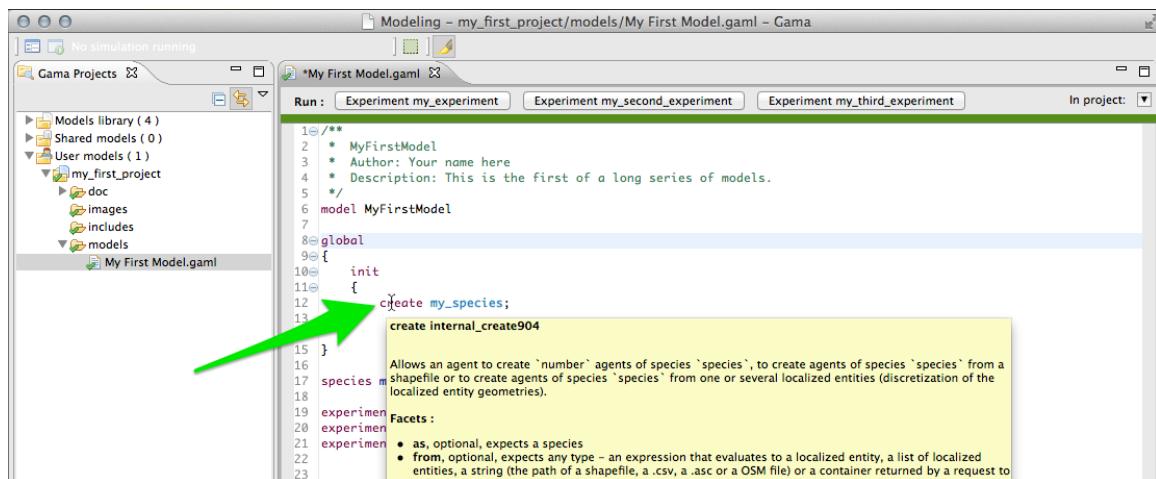


As with the other types of **markers**, information markers unveil their messages when being hovered.



Semantic documentation

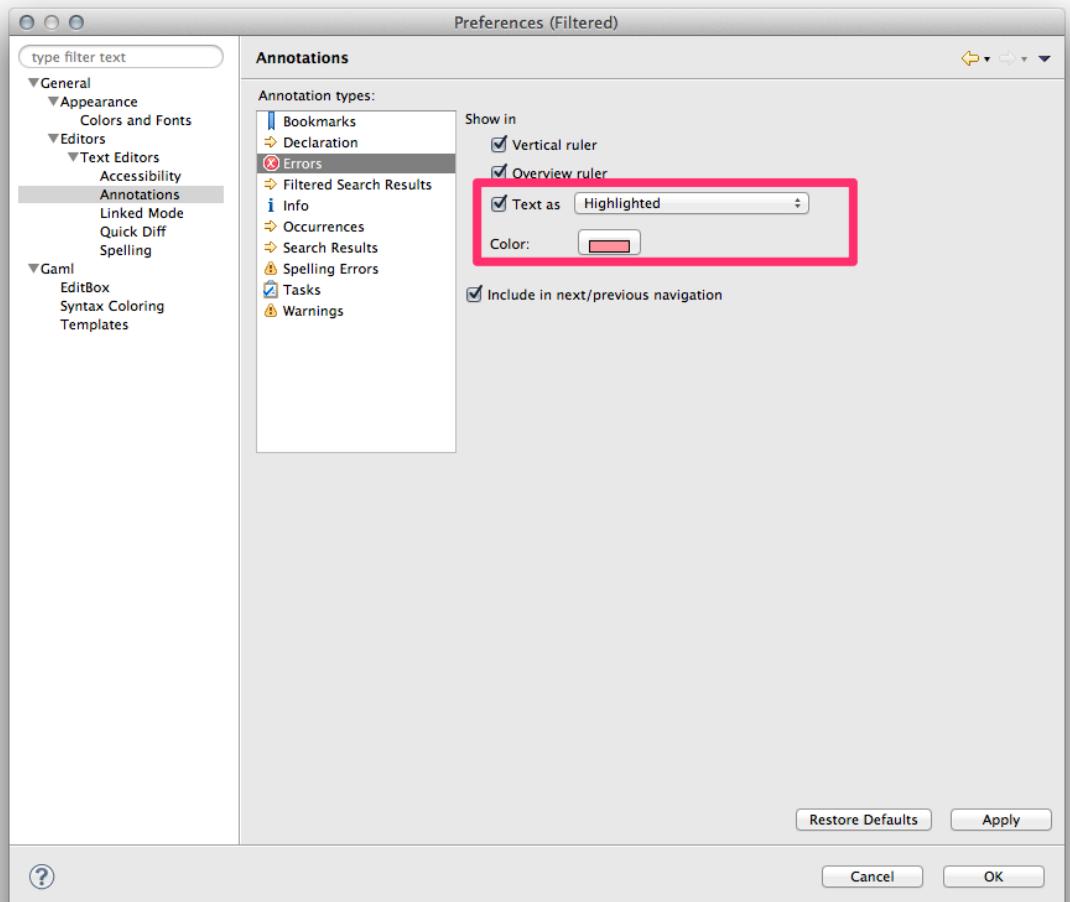
The last type of output of the semantic validation phase consists in a complete documentation of the various elements present in the model, which the user can retrieve by hovering over the different symbols. Note that although the best effort is being made in producing a complete and consistent documentation,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



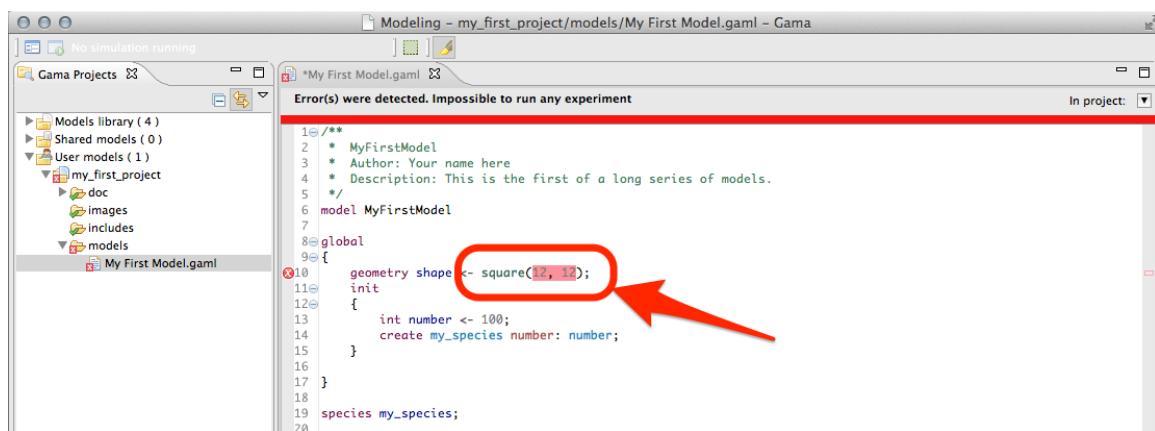
Changing the visual indicators

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.

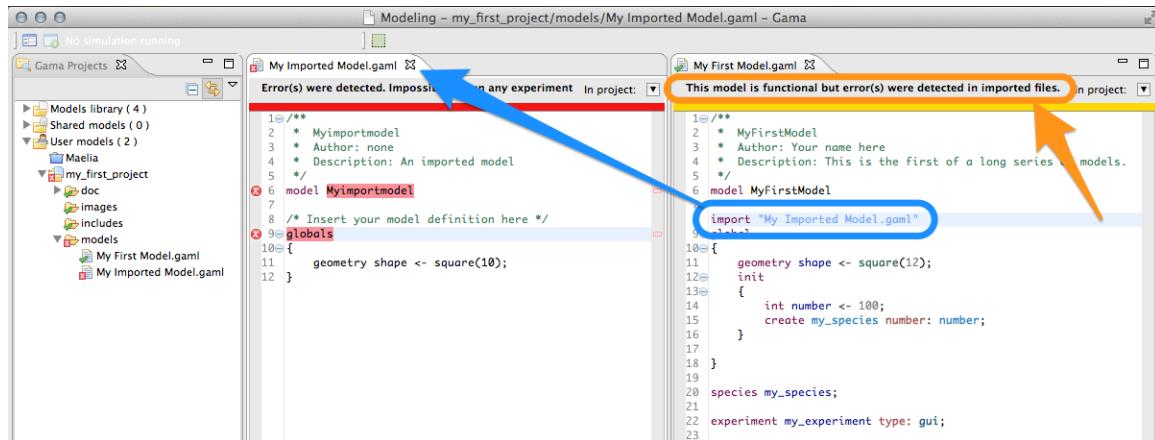


Which will result in the following visual feedback for errors:

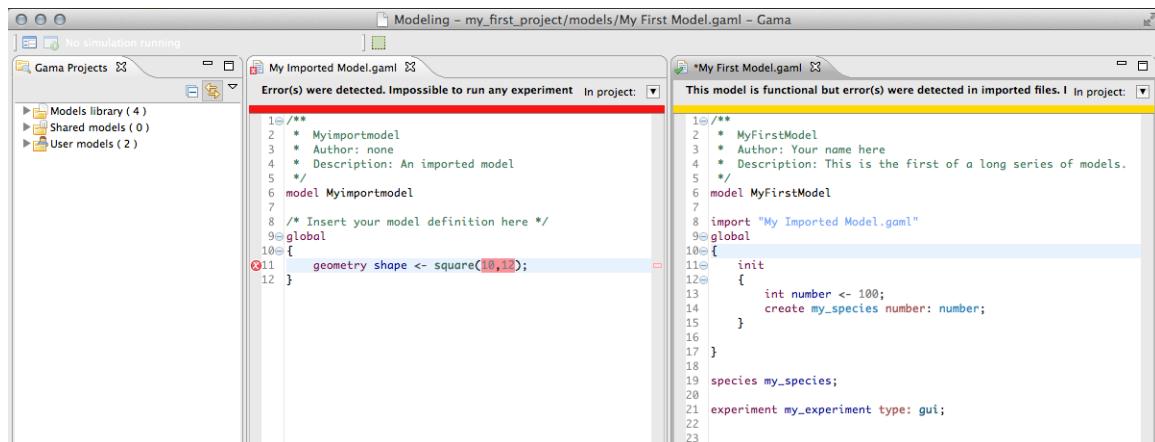


Errors in imported files

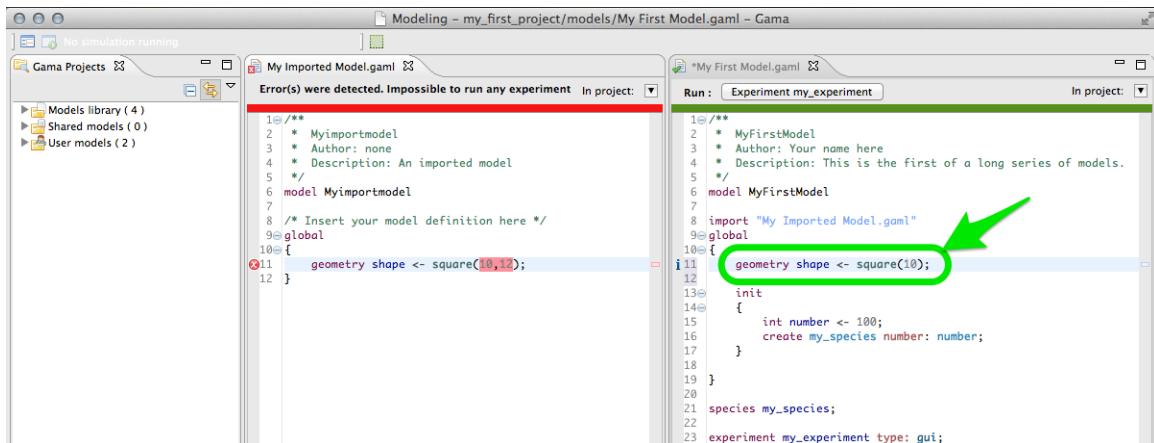
Finally, even if your model has been cleansed of all errors, it may happen that it refuses to launch because it imports another model that cannot be compiled. In the following screenshot, "My First Model.gaml" imports "My Imported Model.gaml", which sports a syntactic error.



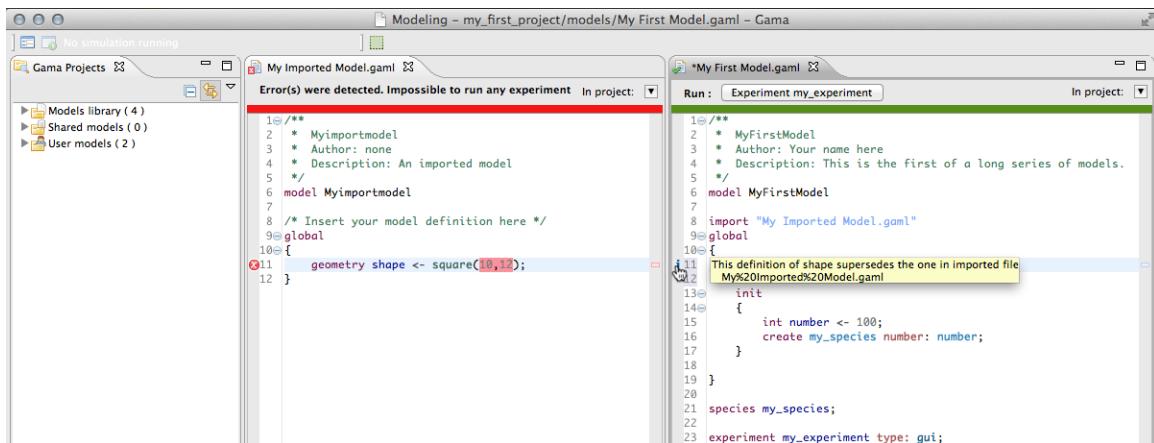
In such a case, the importing model refuses to compile (although it is itself valid) and to propose experiments. There are cases, however, where the same importation can work. Consider the following example, where, this time, "My Imported Model.gaml" sports a semantic error in the definition of the global '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.

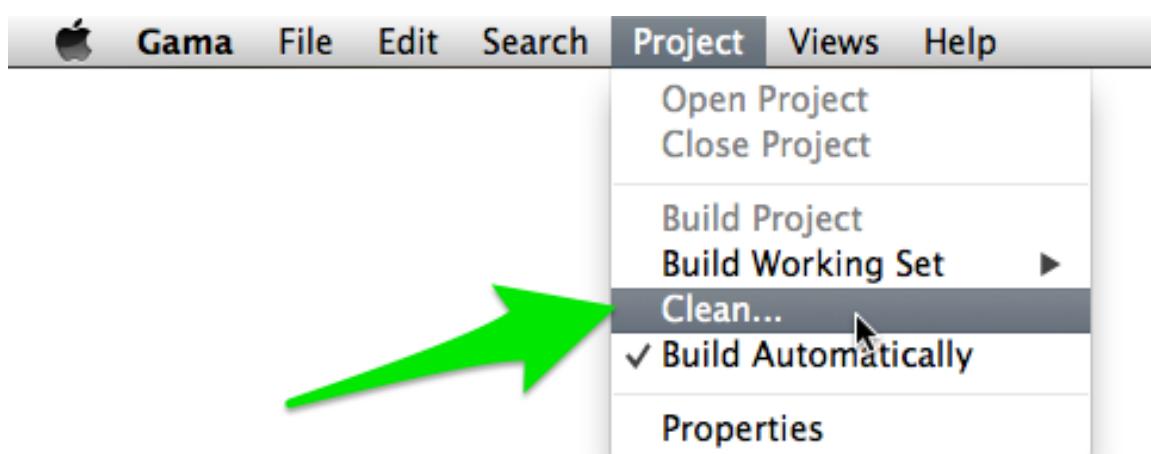


This process is described by the information marker next to the redefinition.

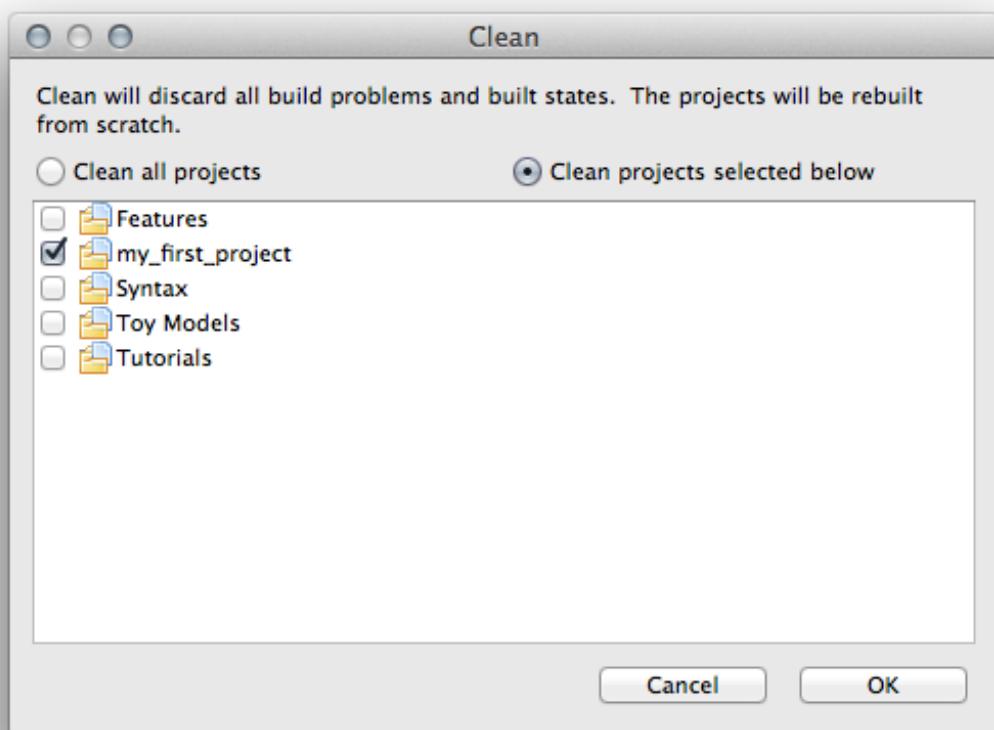


Cleaning models

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, GAMA proposes a menu command called "Clean...", that can be found in the "Project" menu.

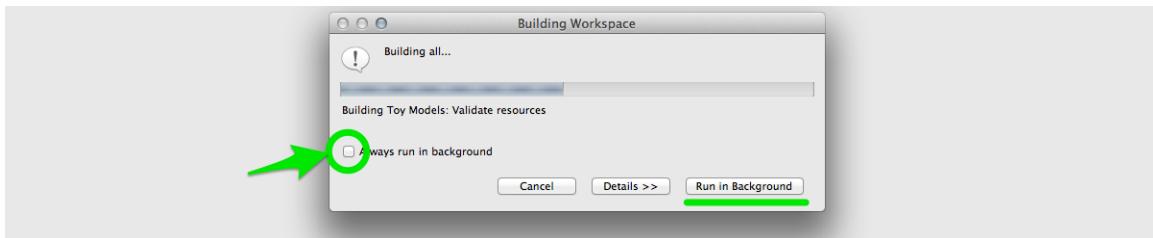


Invoking this command results in GAMA removing all the validation metadata associated with projects and provoking a complete "re-build" of all the projects (*building* here meaning validating the models, essentially). This command can be invoked on the current selected project, or on all projects at once (it is recommended, if you suspect odd behaviors, to do the latter).

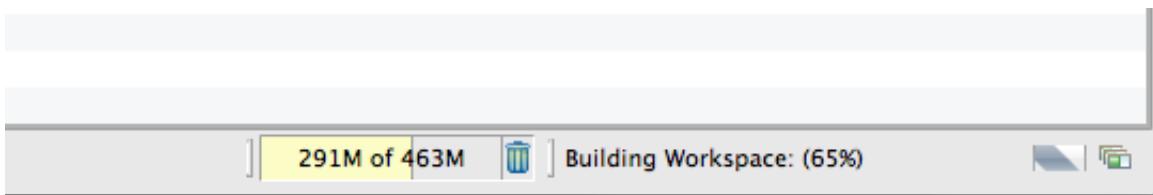


Depending on the size of your projects, the process can take more or less time (a few seconds for the stock version without user projects). Note that GAMA, from time to time, will invoke such complete "re-builds" if it estimates that you have modified too many models. But it doesn't remove the existing

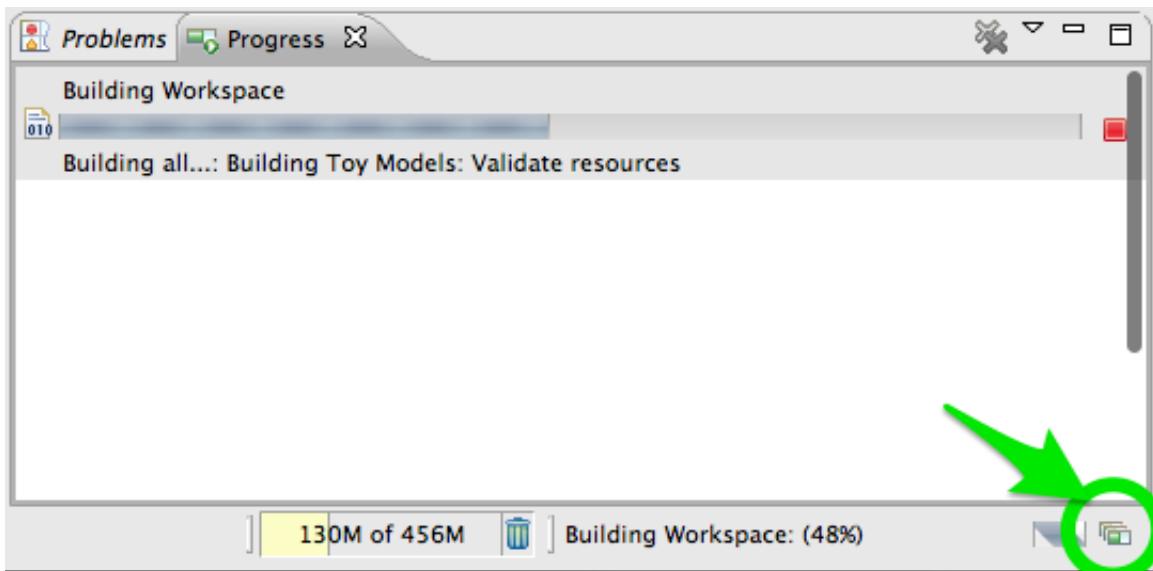
meta-data in this case. This process will display the following dialog, that you can later suppress by enabling it to run in the background.



When running in the background, GAMA will report the progress of the building process in the bottom right-hand corner of the window. The small icon next to it enables to follow it more accurately (while still running in background)



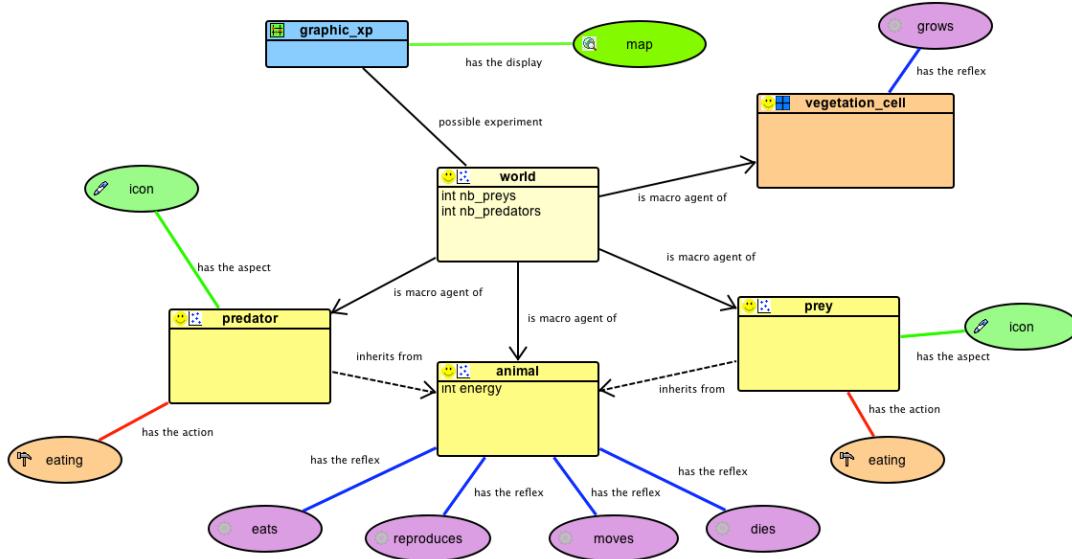
This will open a new view, called *Progress* in which other background processes (workspace management, memory management) will also report their progresses. This view can be opened and closed at will without provoking any trouble.



3.3 Graphical Editor

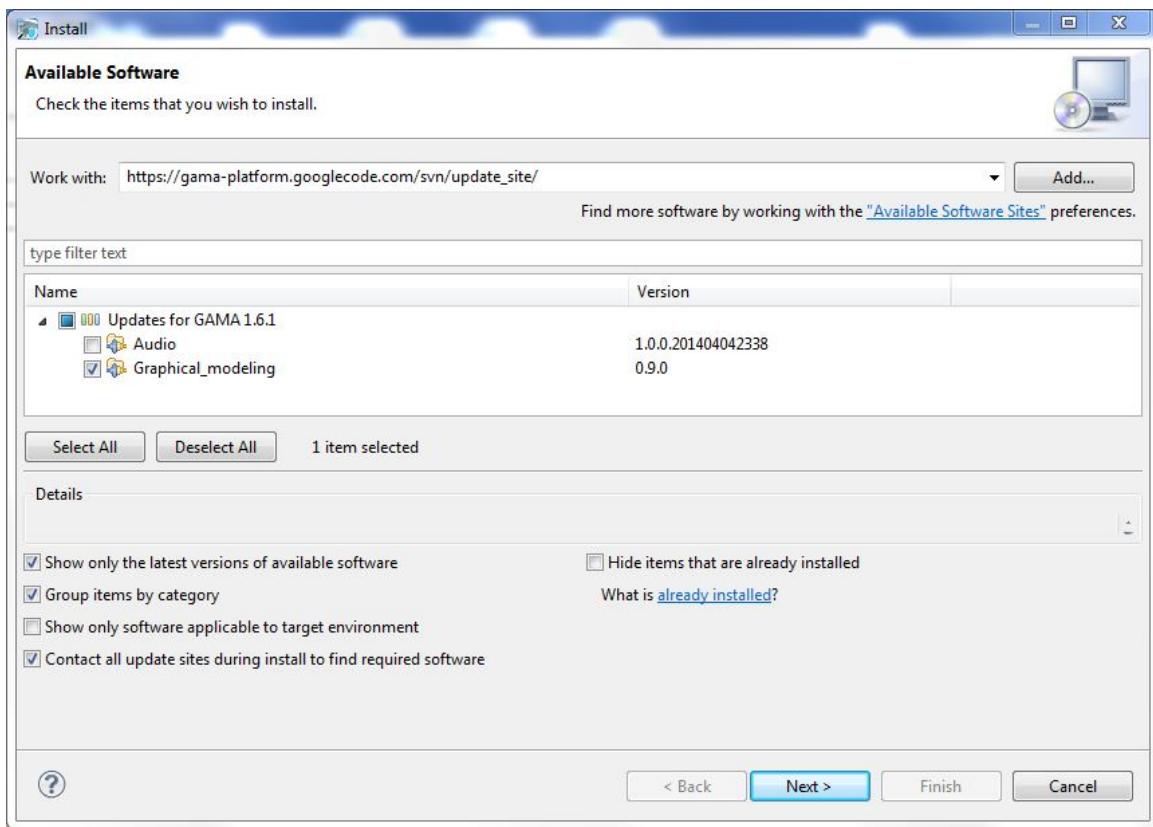
The Graphical Editor

The graphical editor that allow to build diagram (gadl files) is based on the [Graphiti Eclipse plugin](#). It allows to define a GAMA model through a graphical interface. It also allows as well to produce a graphical model (diagram) from a gaml model.



Installing the graphical editor

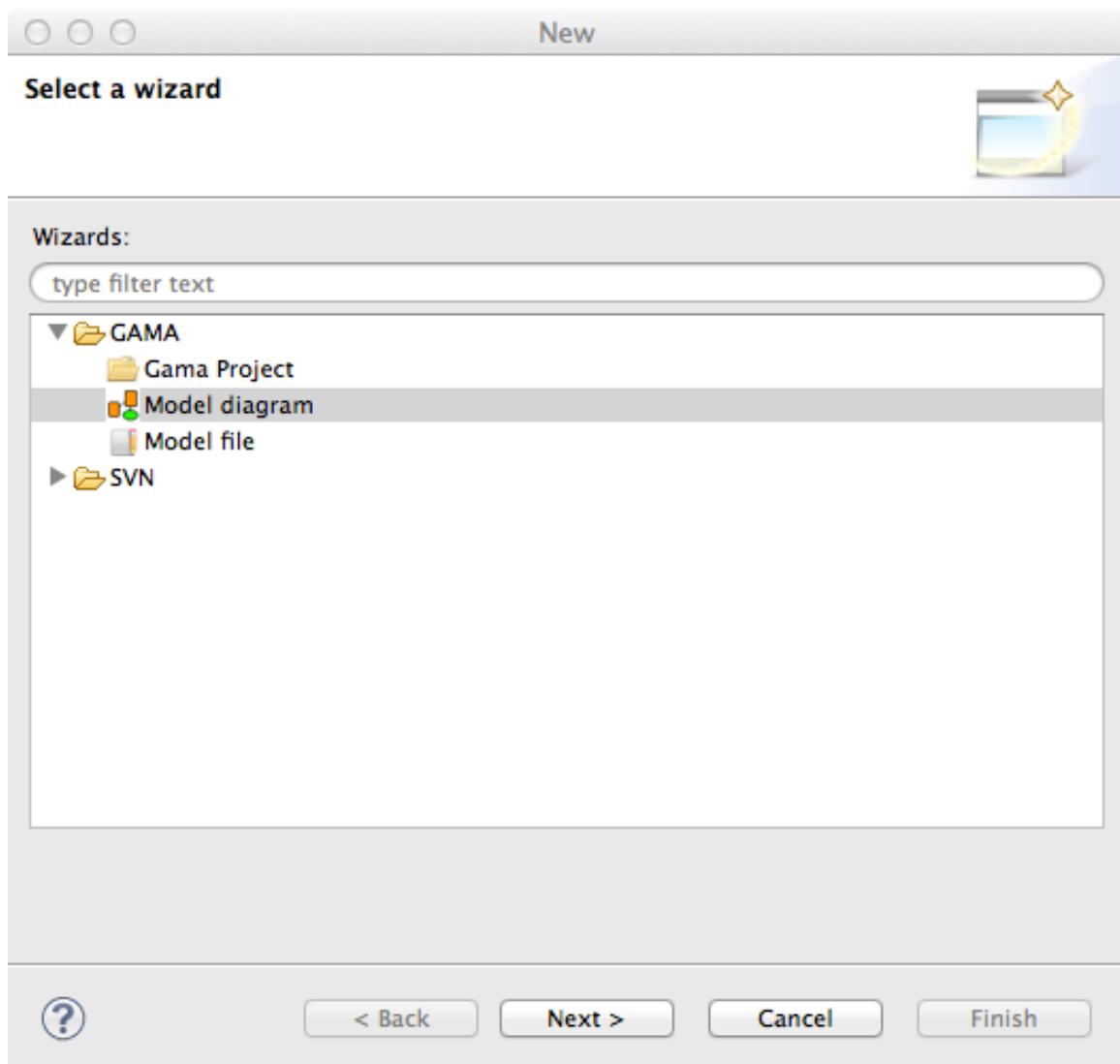
Using the graphical editor requires to install the graphical modeling plug-in. See [here](#) for information about plug-ins and their installation. The graphical editor plug-in is called **Graphical_modeling** and is directly available from GAMA update site https://gama-platform.googlecode.com/svn/update_site/.



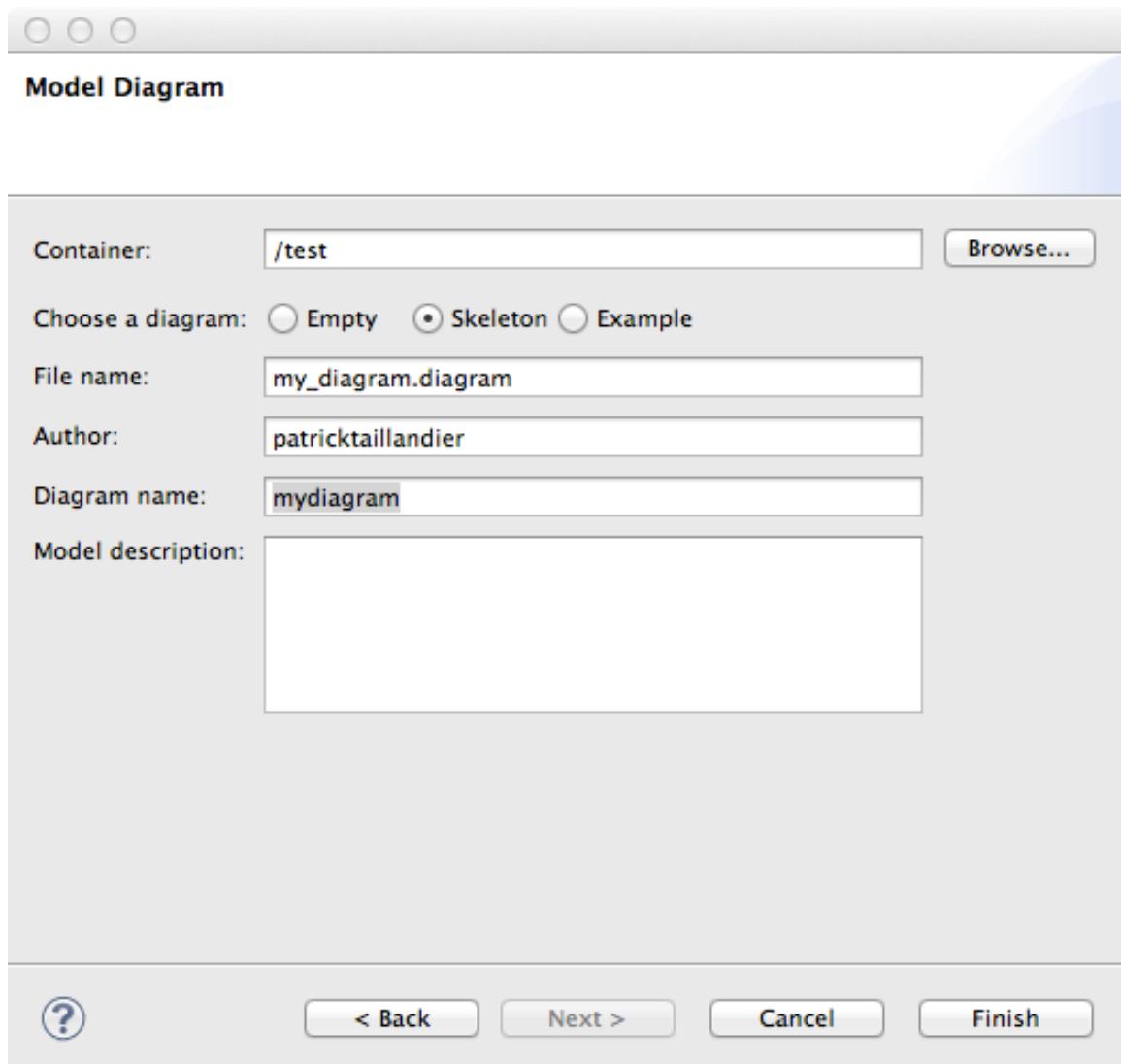
Note that the graphical editor is still under development. Updates of the plug-in will be add to the GAMA website. After installing the plug-in (and periodically), check for updates for this plug-in: in the "Help" menu, choose "Check for Updates" and install the proposed updates for the graphical modeling plug-in.

Creating a first model

A new diagram can be created in a new GAMA project. First, right click on a project, then select "New" on the contextual menu. In the New Wizard, select "GAMA -> Model Diagram", then "Next>"



In the next Wizard dialog, select the type of diagram (Empty, Skeleton or Example) then the name of the file and the author.



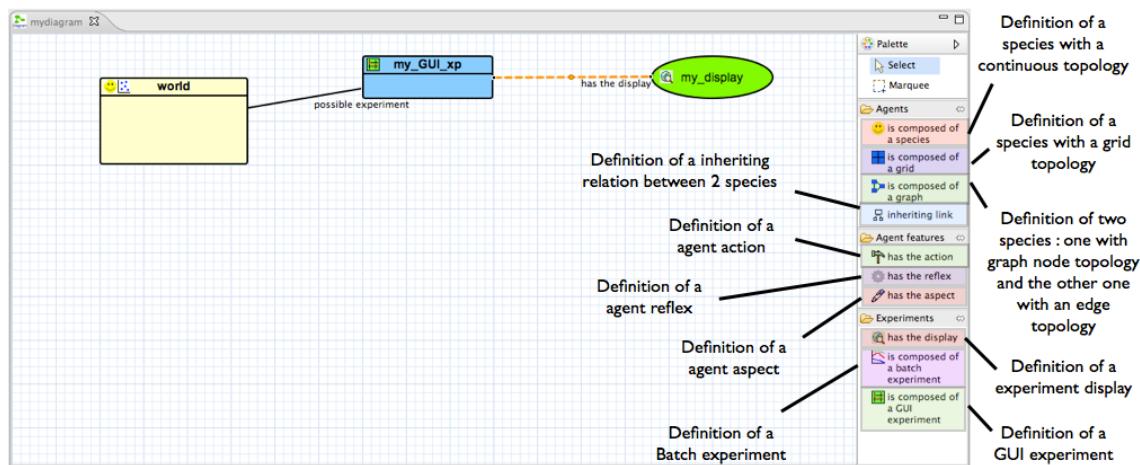
Skeleton and Example diagram types allow to add to the diagram some basic features.

Status of models in editors

Similarly to GAML editor, the graphical editor proposes a live display of errors and model statuses. A graphical model can actually be in three different states, which are visually accessible above the editing area: **Functional** (orange color), **Experimentable** (green color) and **[InError]** (red color). See [CompilingModels161 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 occurs when the file contains errors (syntactic or semantic ones). Reaching the **Experimentable** state requires that all errors are eliminated and that at least one experiment is defined in the model. The experiment is immediately displayed as a button in the toolbar, and clicking on it will allow to launch this experiment on your model. 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.

Diagram definition framework

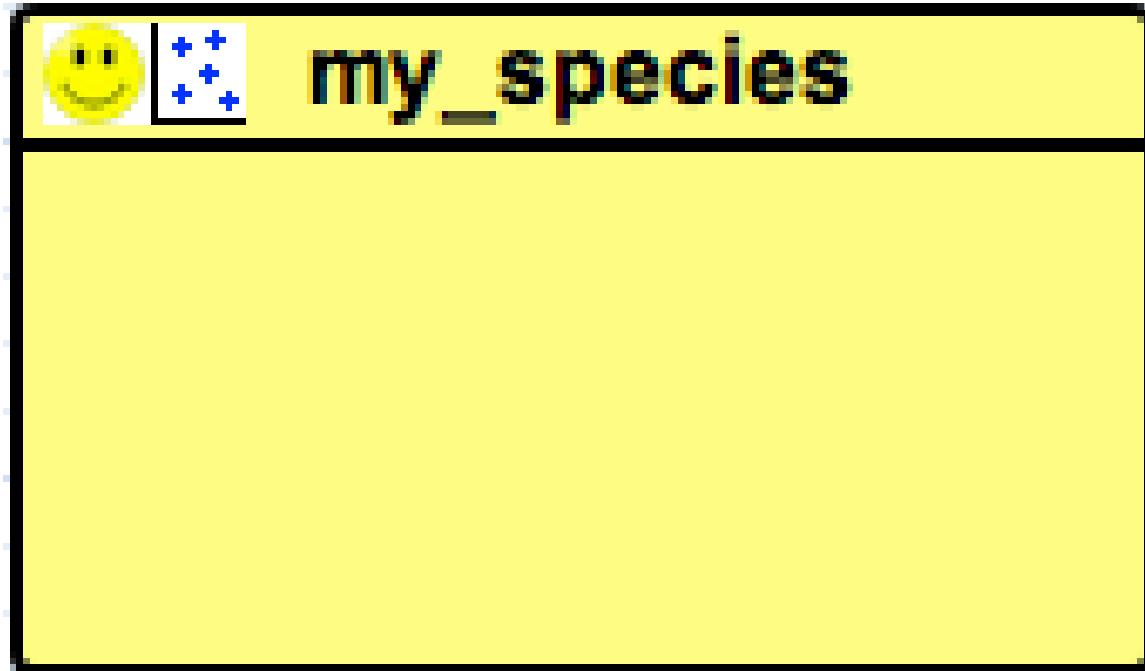
The following figure presents the editing framework:



Features

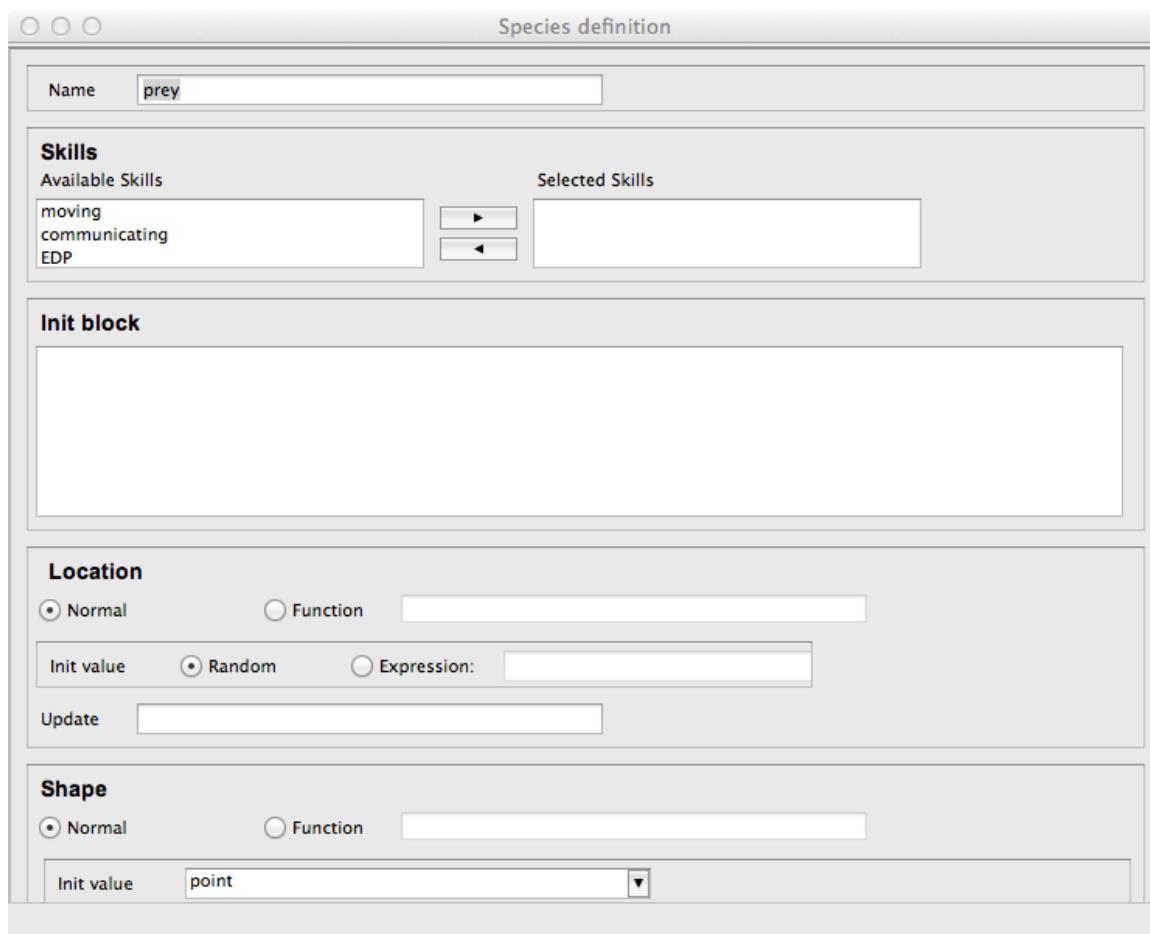
agents

species



The species feature allows to define a species with a continuous topology. A species is always a micro-species of another species. The top level (macro-species of all species) is the world species.

- **source** : a species (macro-species)
- **target** : -



Species definition

Shape

Normal Function

Init value

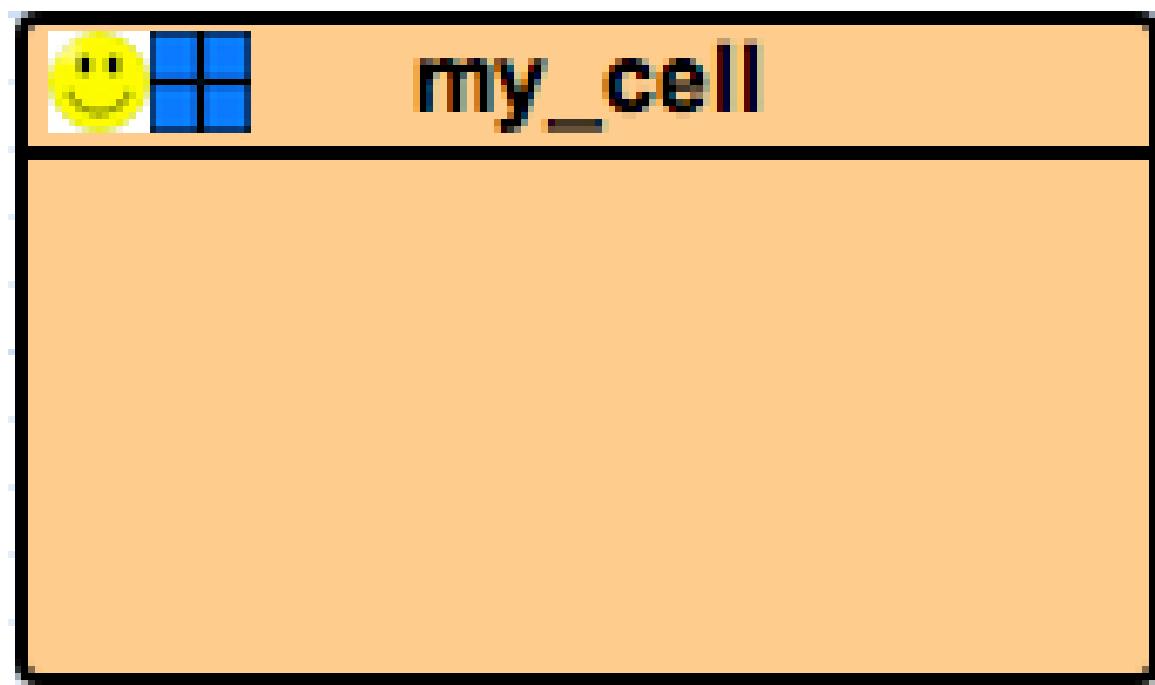
Update

Variables

Name	Type	init value	update	function	min	max
myCell	vegetation_cell	one_of (list (ve...				
max_energy	float	prey_max_energy				
max_transfert	float	prey_max_tran...				
size	float	2.0				
color	rgb	rgb ('blue')				
energy_consum	float	prey_energy_co...				
energy	float	*(/(rnd (1000),...	-(energy,energ...			max_energy

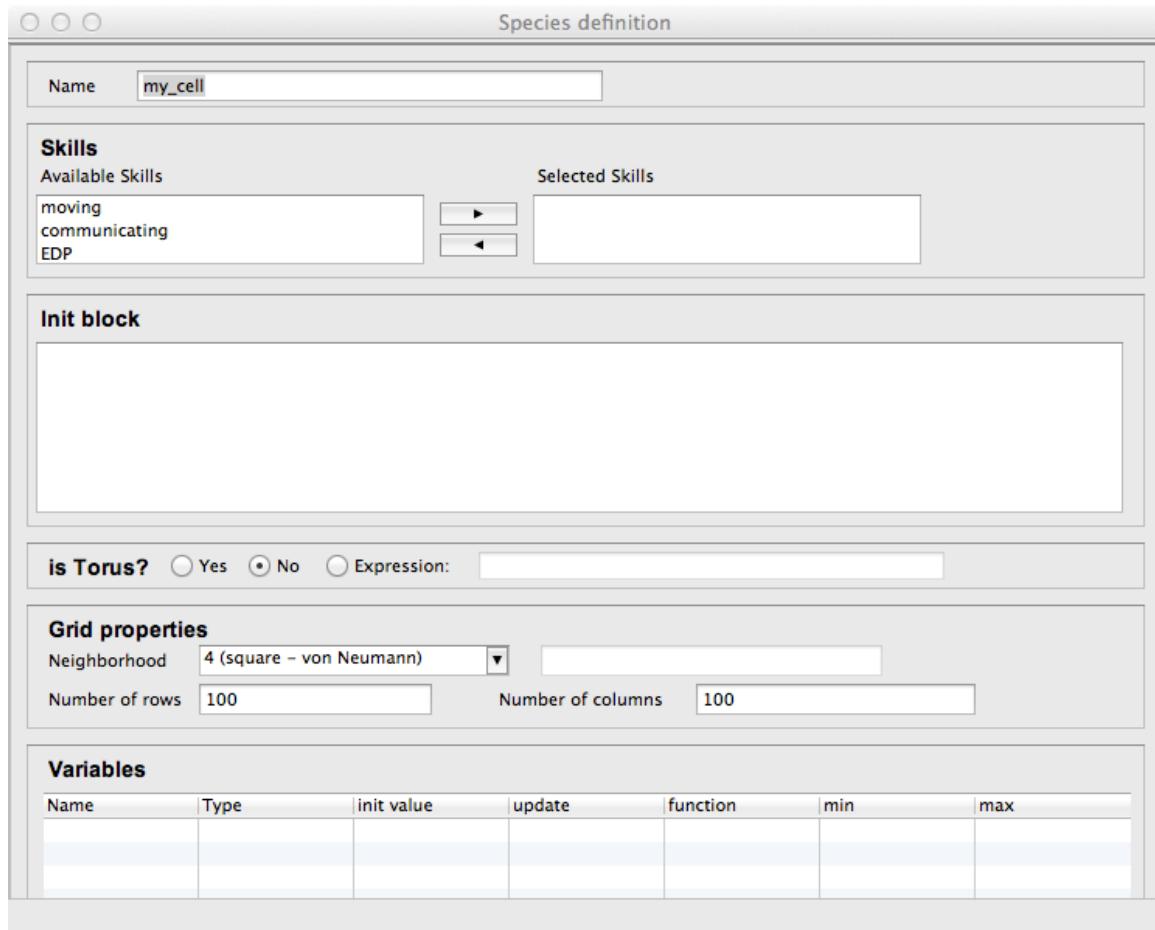
Reflex order

grid



The grid feature allows to define a [Species151 species] with a [Sections151 grid topology]. A grid is always a micro-species of another species.

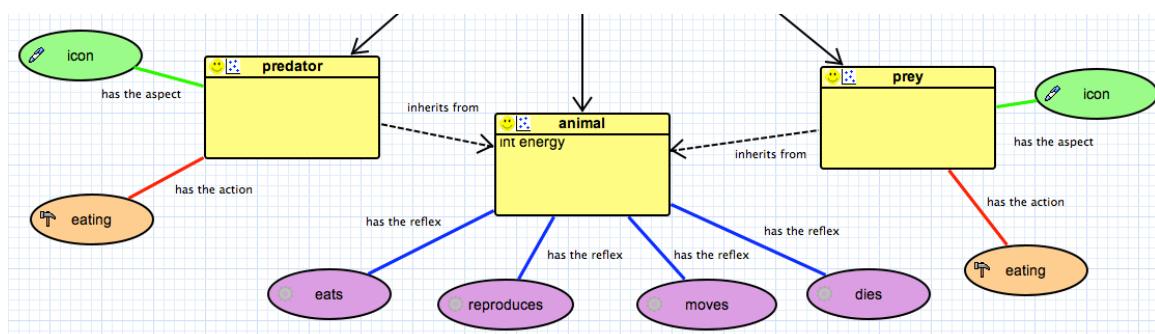
- **source** : a species (macro-species)
- **target** : -



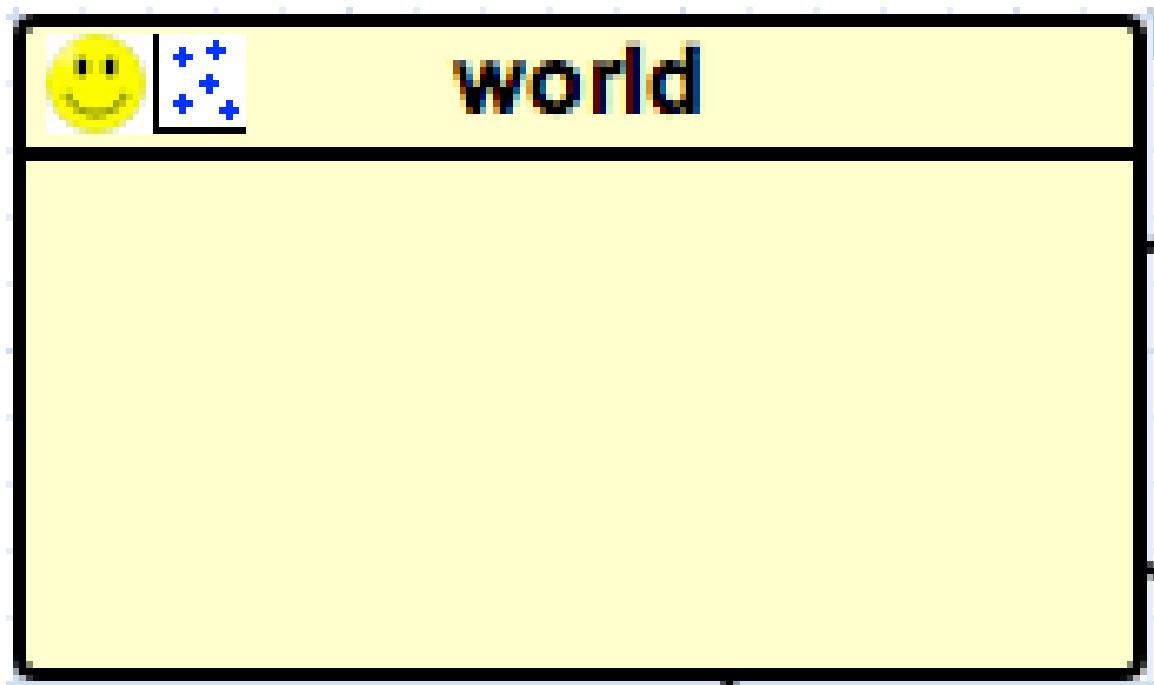
Inheriting link

The inheriting link feature allows to define an inheriting link between two species.

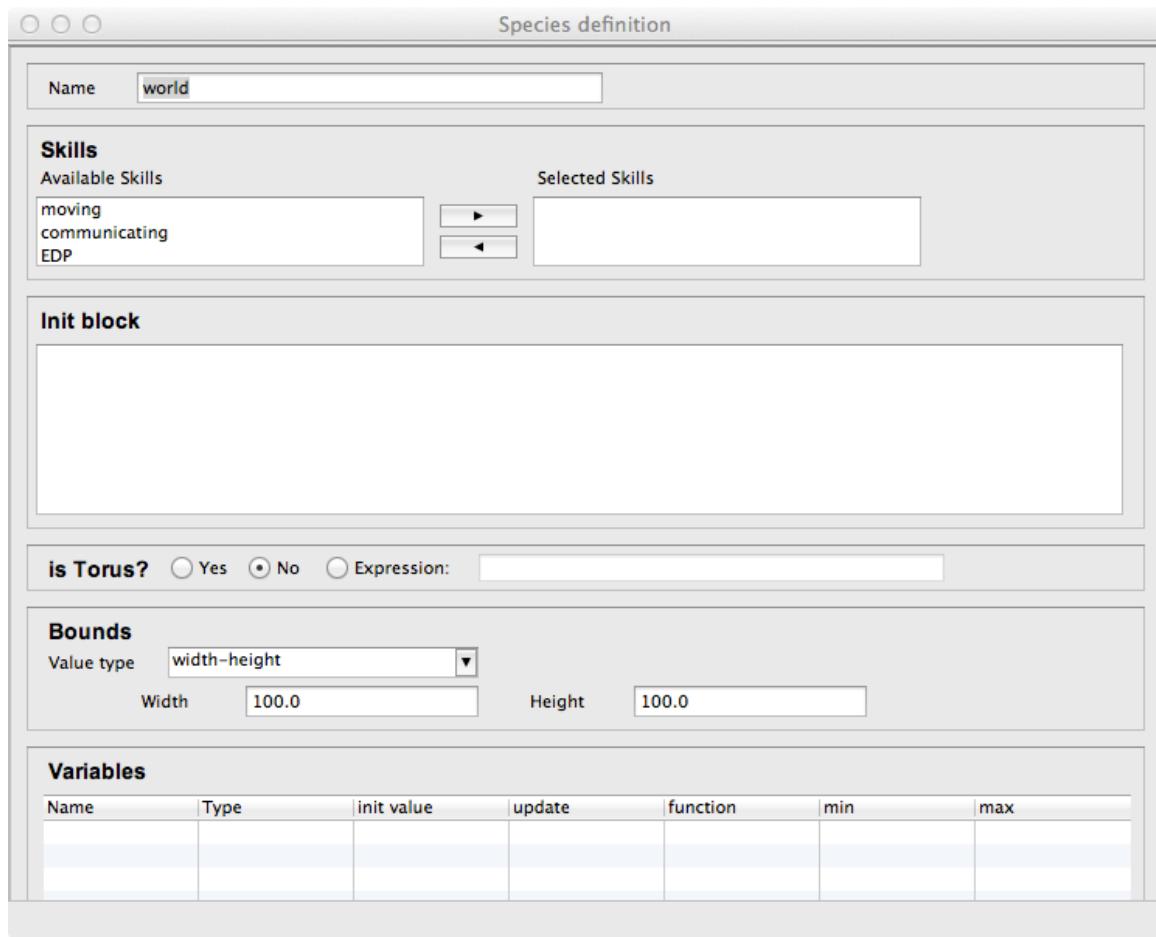
- **source** : a species (parent)
- **target** : a species (child)



world



When a model is created, a world species is always defined. It represent the global part of the model. The world species, which is unique, is the top level species. All other species are micro-species of the world species.



agent features

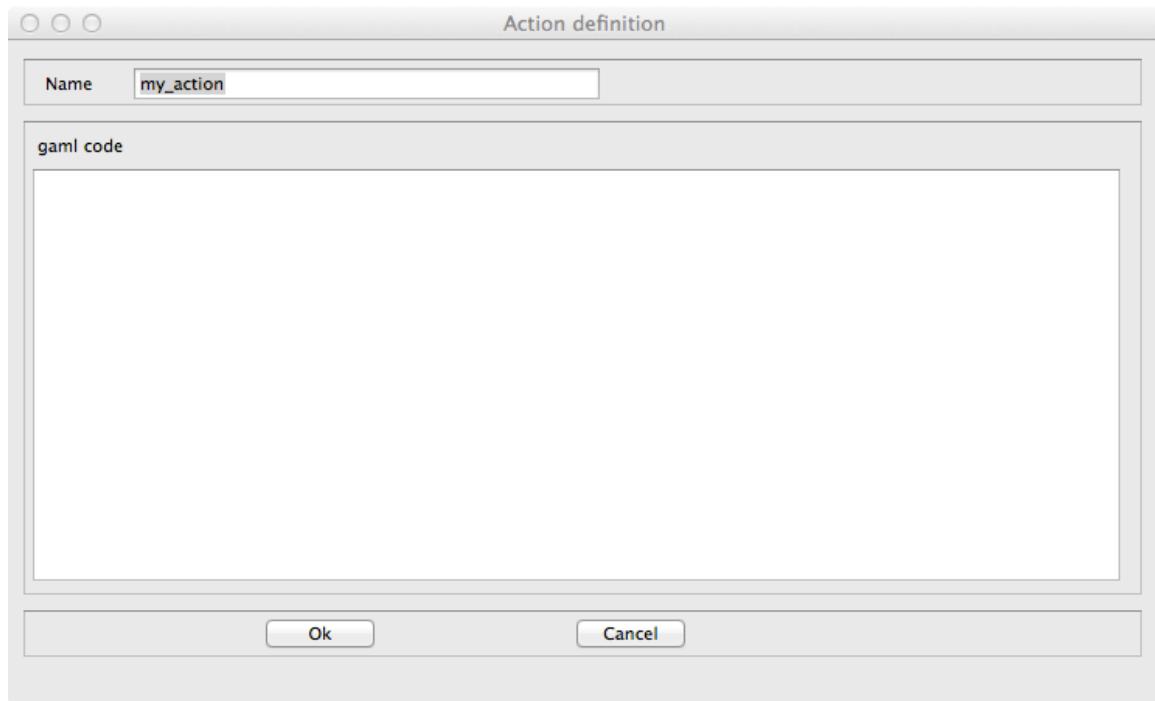
action



The action feature allows to define an action for a species.

- **source** : a species (owner of the action)

- **target** :-

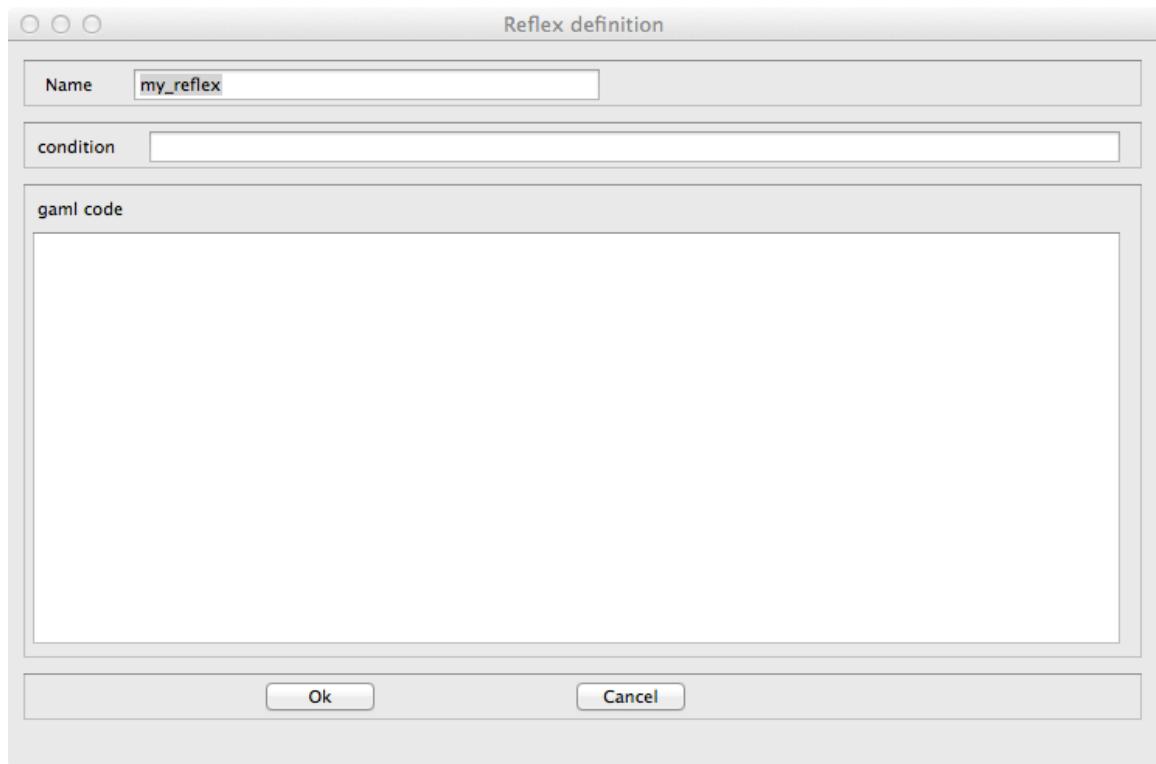


reflex



The reflex feature allows to define a reflex for a species.

- **source** : a species (owner of the reflex)
- **target** :-

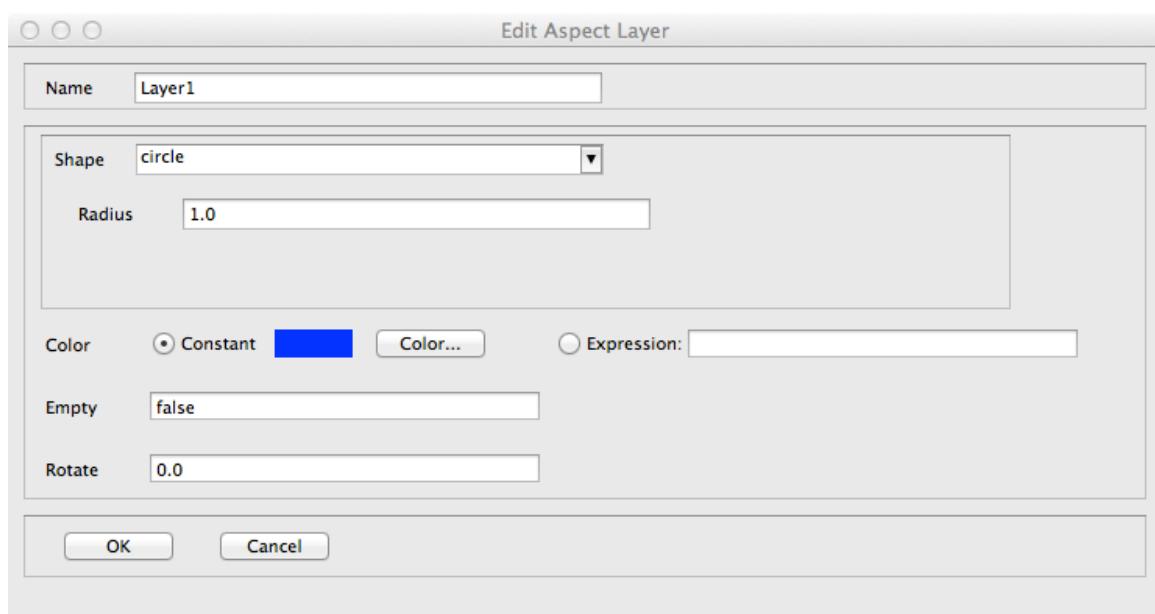
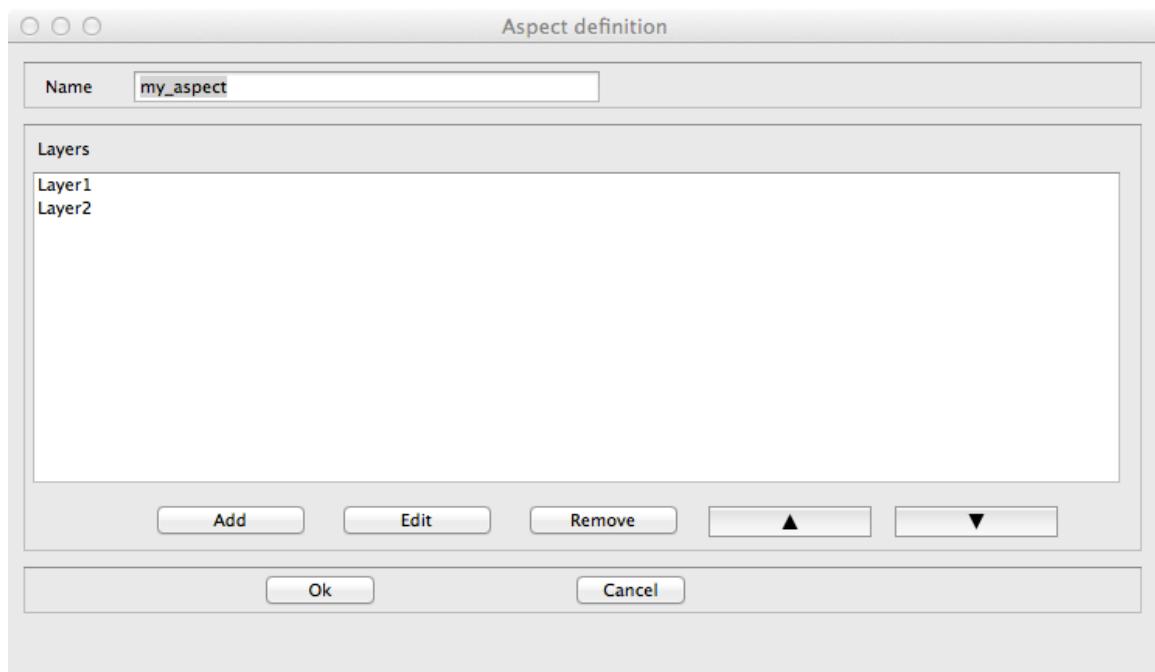


aspect



The aspect feature allows to define an aspect for a species.

- **source** : a species (owner of the aspect)
- **target** :-



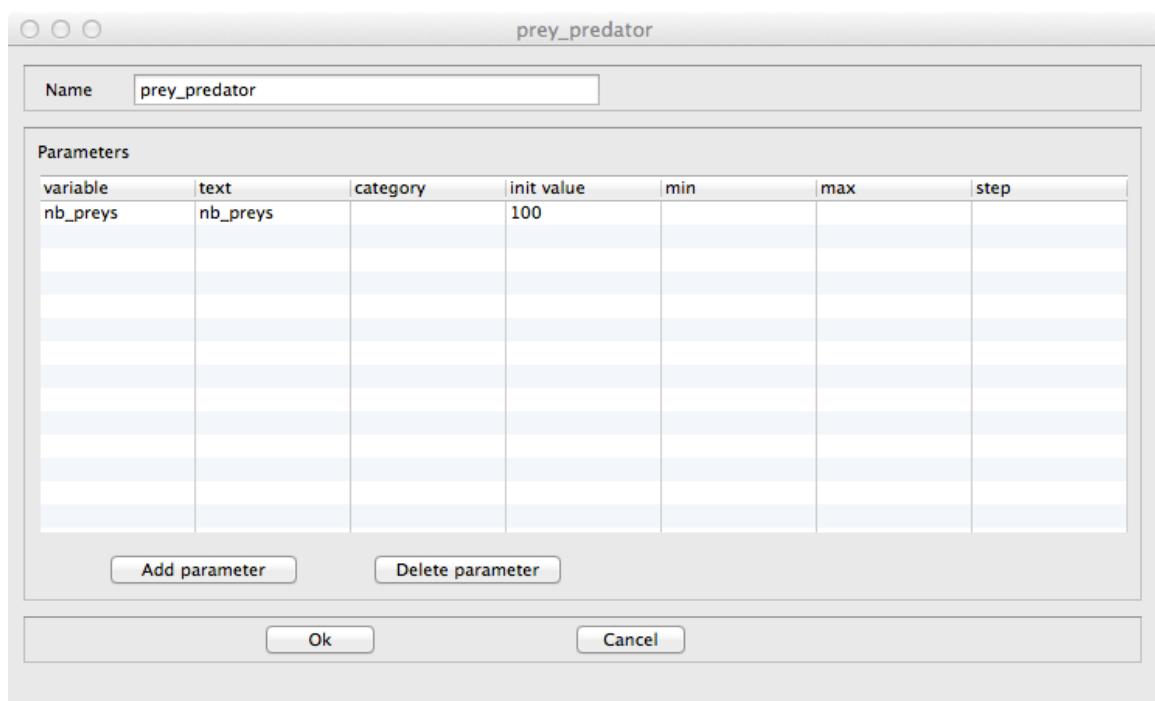
experiment

GUI experiment

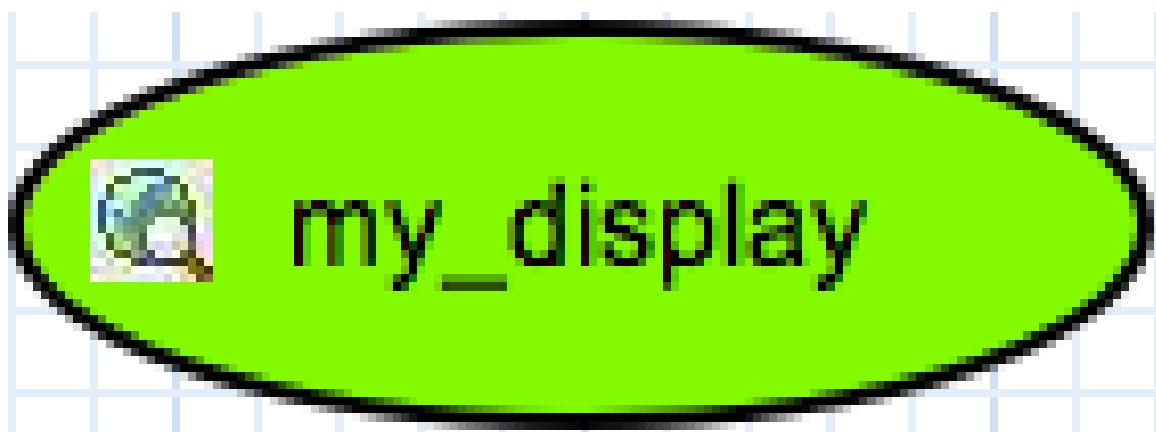


The GUI Experiment feature allows to define a GUI experiment.

- **source** : world species
- **target** :-

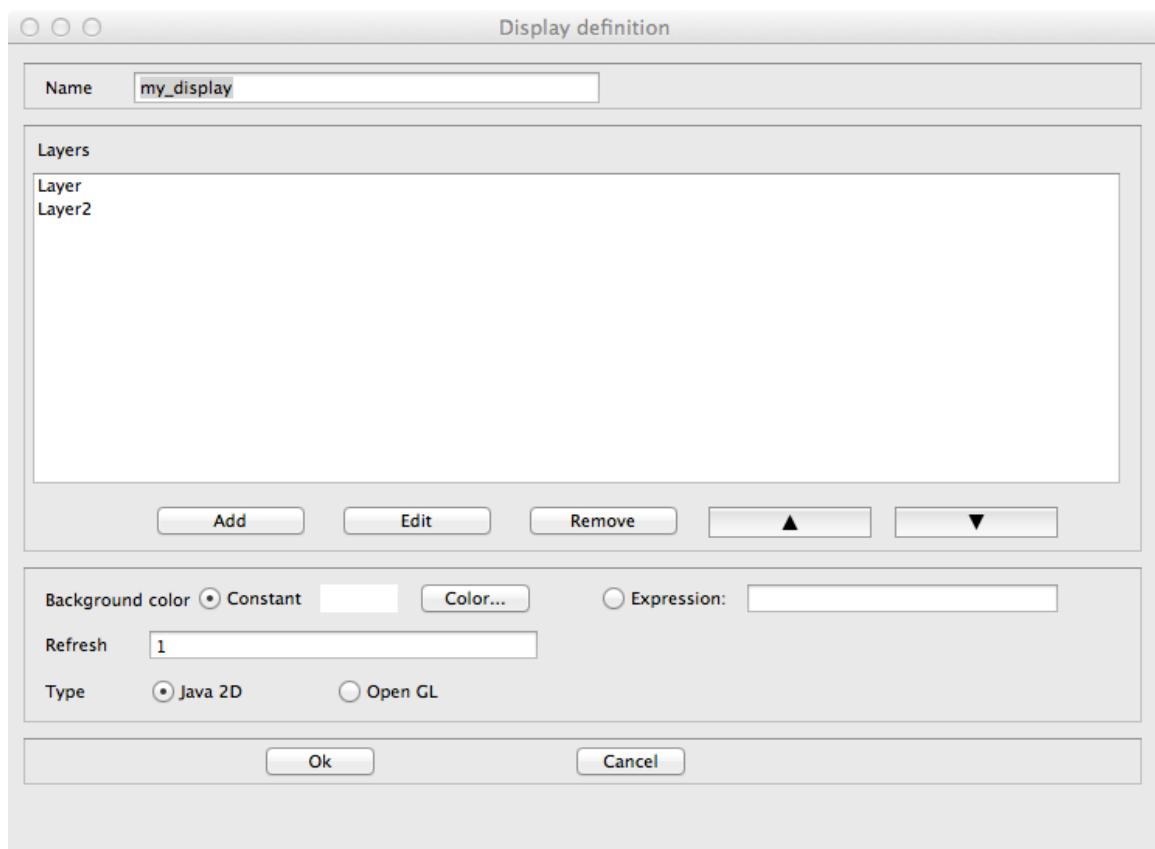


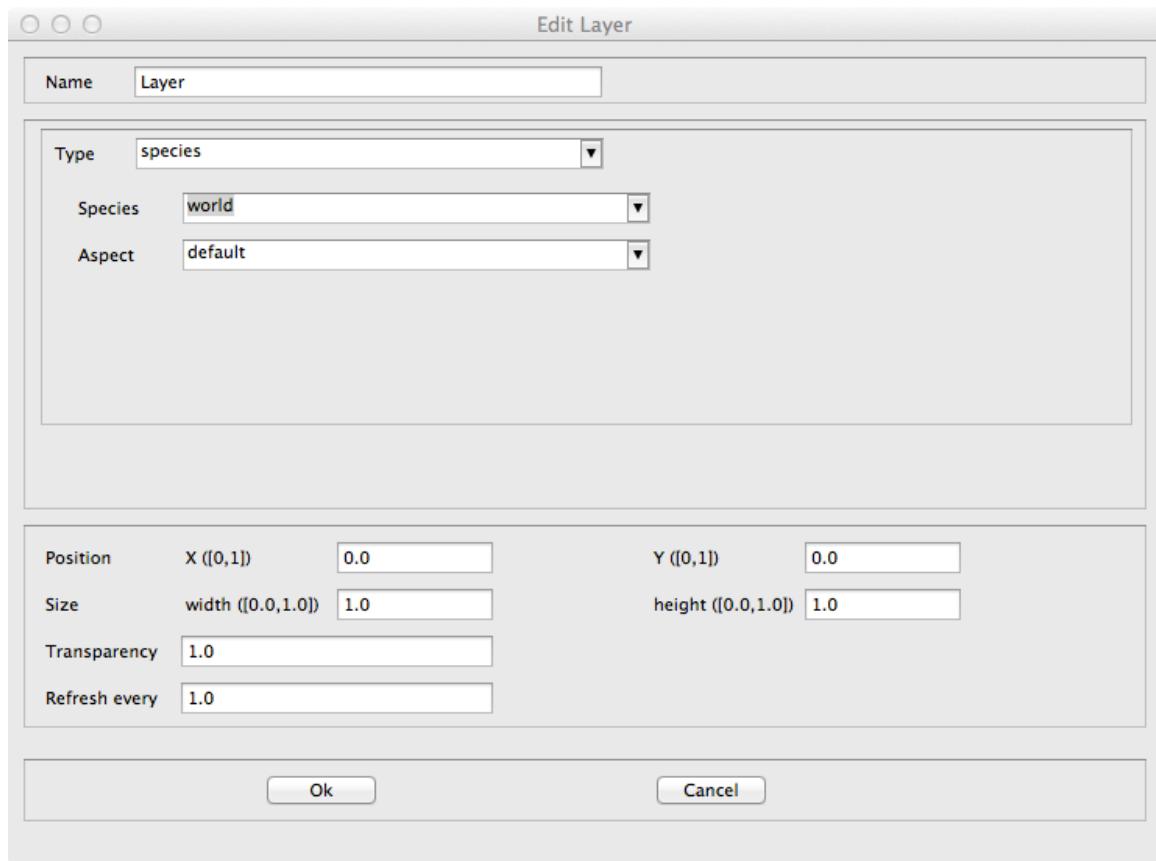
display



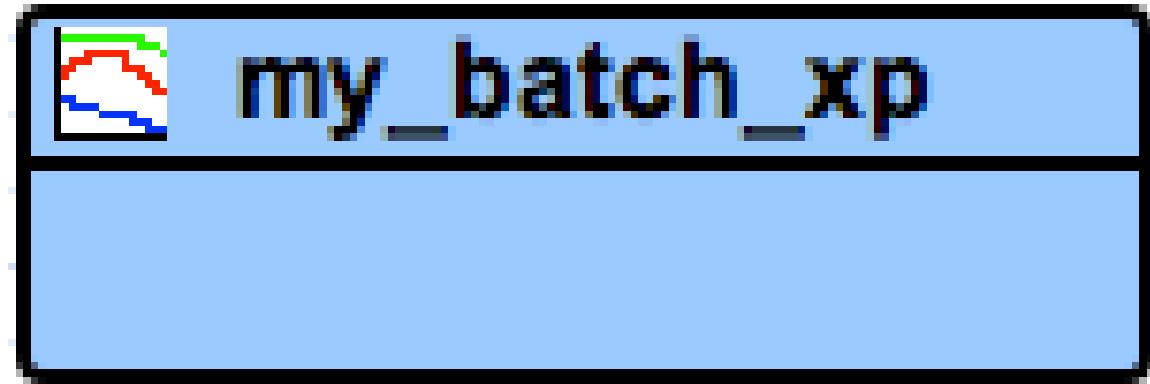
The display feature allows to define a display.

- **source** : GUI experiment
- **target** :-





batch experiment



The Batch Experiment feature allows to define a Batch experiment.

- **source** : world species
- **target** : -

Pictogram color modification

It is possible to change the color of a pictogram.

- Right click on a pictogram, then select the "Chance the color".
-

GAML Model generation

It is possible to automatically generate a Gaml model from a diagram.

- Right click on the graphical framework (where the diagram is defined), then select the "Generate Gaml model".

A new GAML model with the same name as the diagram is created (and open).

4. Running Experiments

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.
2. The second way, detailed on this [page](#), allows to automatically launch an experiment when opening GAMA, subsequently using the same [user interface](#).
3. 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.
2. The second is intended to be used when demonstrating or experimenting a single model.
3. The last one is useful when running large sets of simulations, especially over networks or grids of computers.

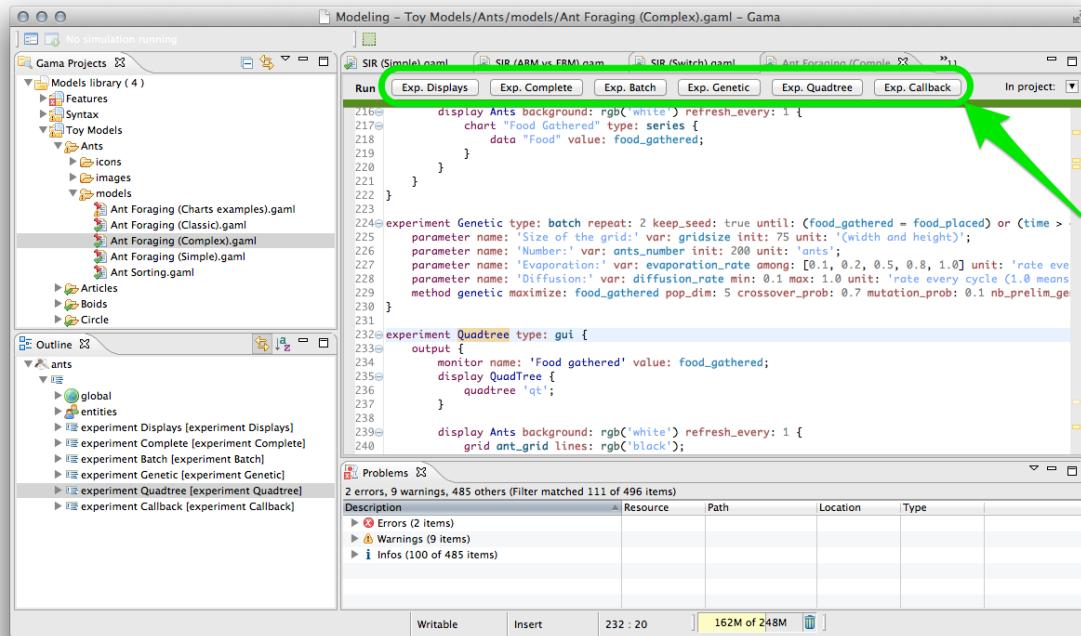
4.1 Launching Experiments

Launching Experiments from the User Interface

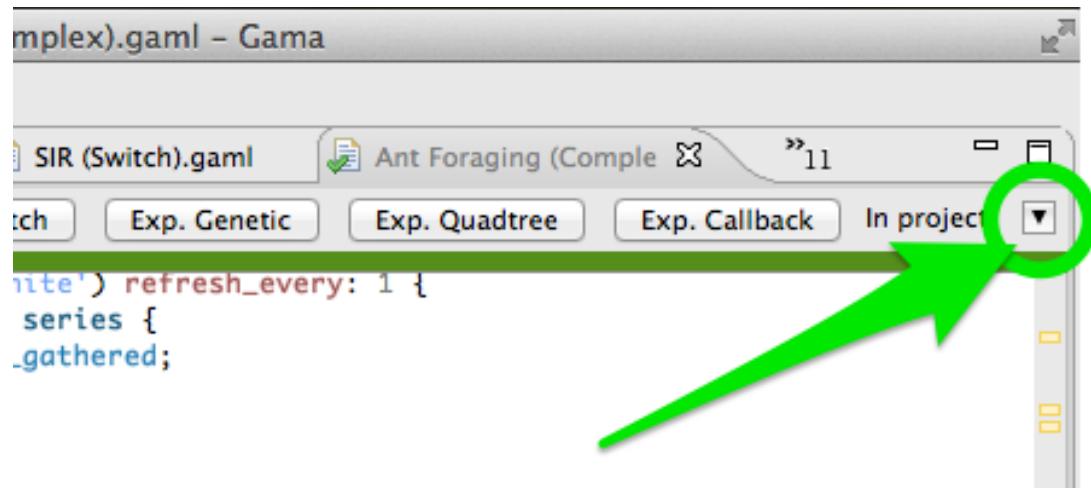
GAMA supports multiple ways of launching experiments from within the Modeling Perspective, in editors or in the [navigator](#).

From an Editor

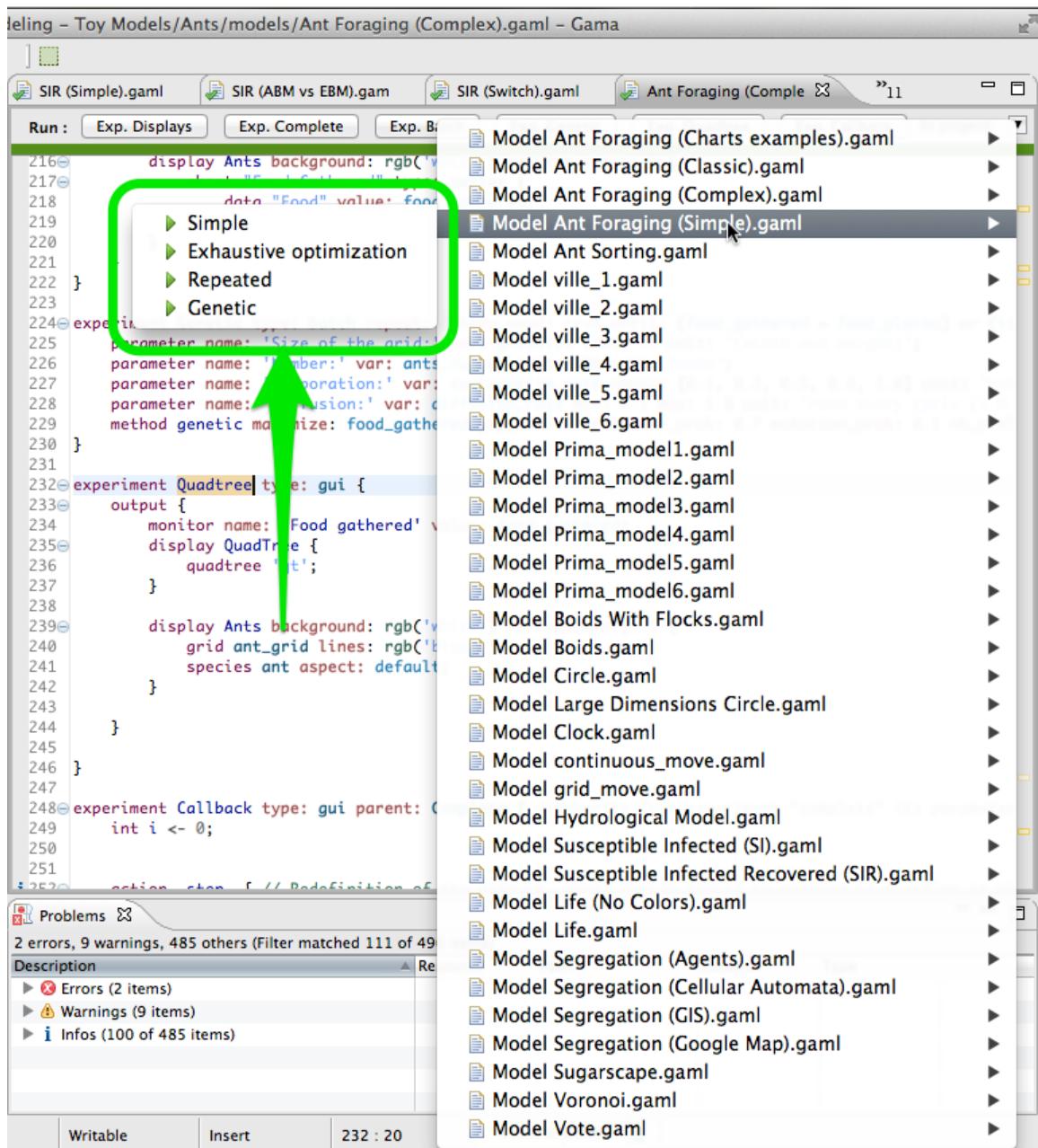
As already mentioned on [this page](#), GAML editors will provide the easiest way to launch experiments. Whenever a model that contains the definition of experiments is validated, these experiments will appear as distinct buttons, in the order in which they are defined in the file, in the header ribbon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



In case the currently edited file does not define any experiment, but if other files *in the same project* define some, it is possible to access them directly from the editor using the small "In project:" drop-down arrow on the upper-right corner of the editor.



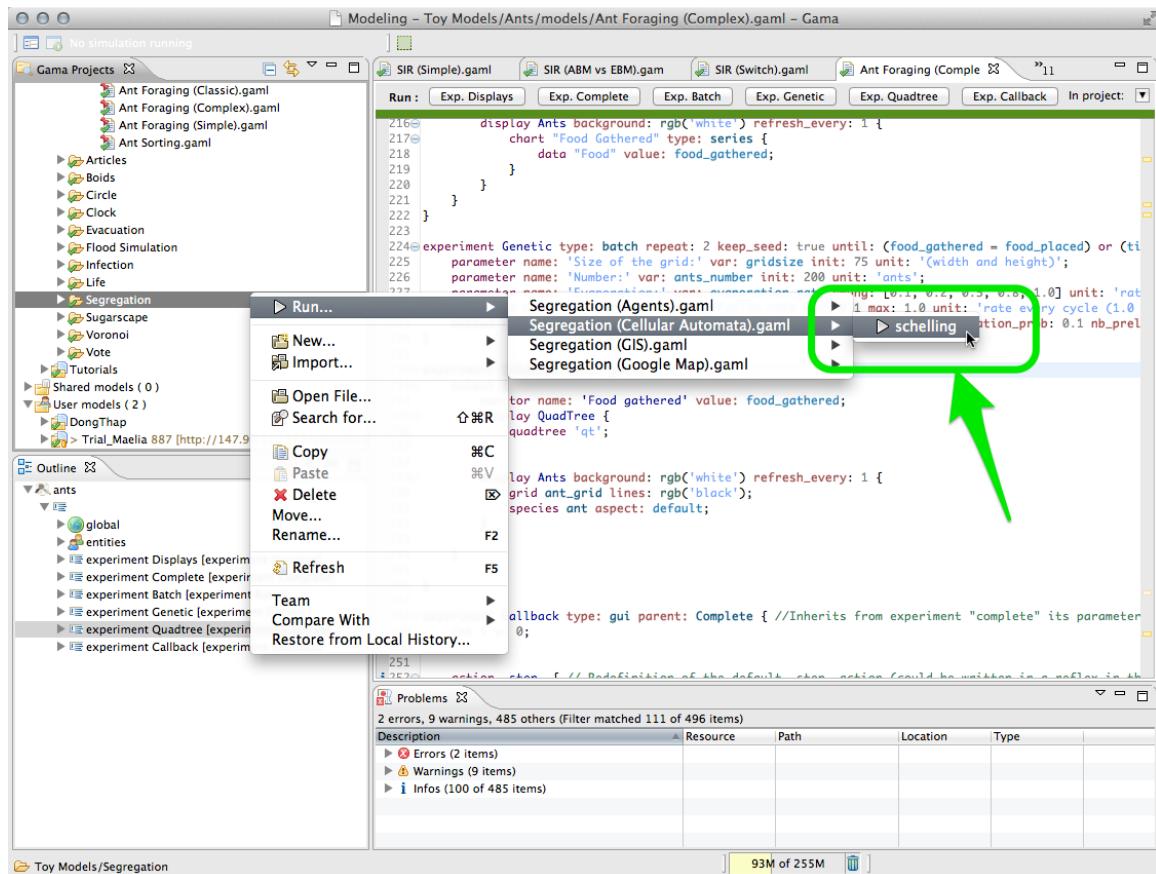
This menu gives access to all the model files currently defined with experiments in the project and, for each, to its different experiments. Selecting one will have the same effect as clicking on the button in the editor of this file.



From the Navigator

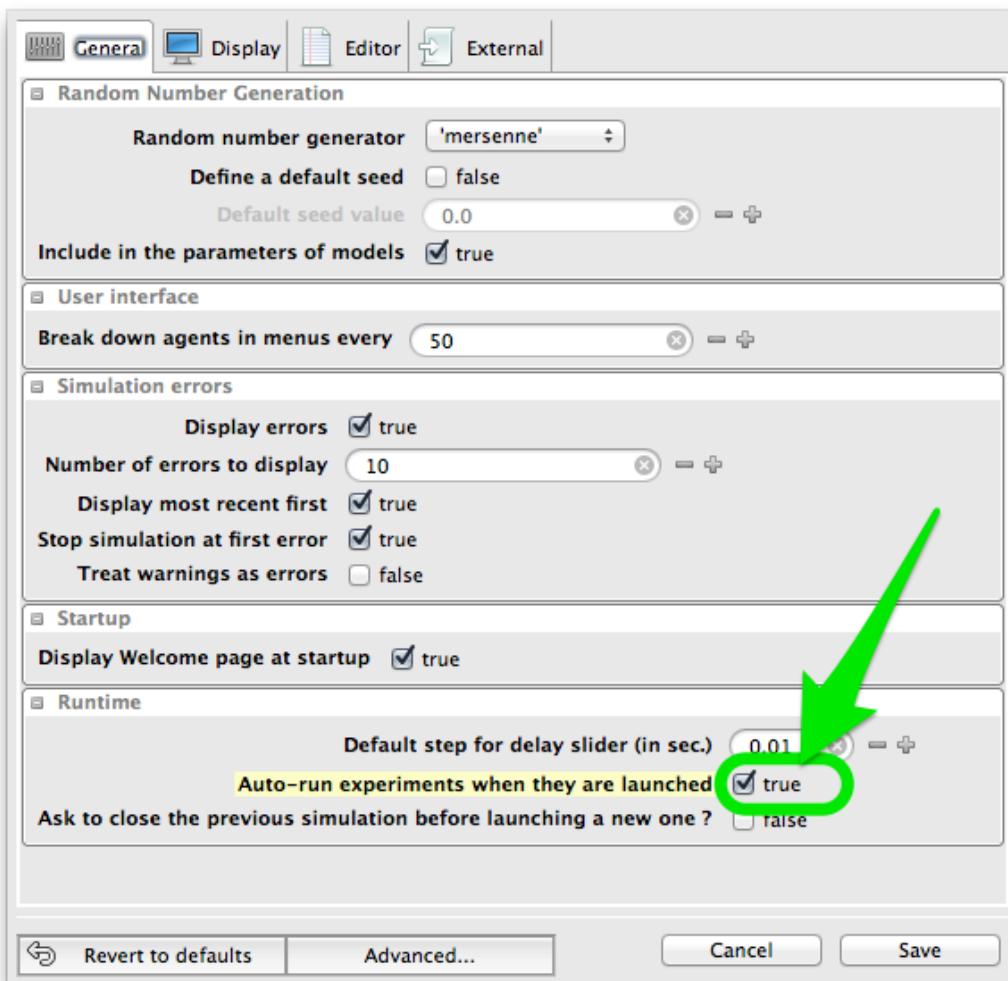
A same mechanism is implemented in the [Navigator](#). Right-clicking on a category, on a project, on a sub-directory of a project, or on a model file will enable the command named "Run..." in the contextual menu. Depending on the object(s) being selected, this command will give you access to:

1. The experiments defined in the model file(s) selected
2. The experiments defined in the model file(s) present in the category(s), project(s) or folder(s) selected.



Running Experiments Automatically

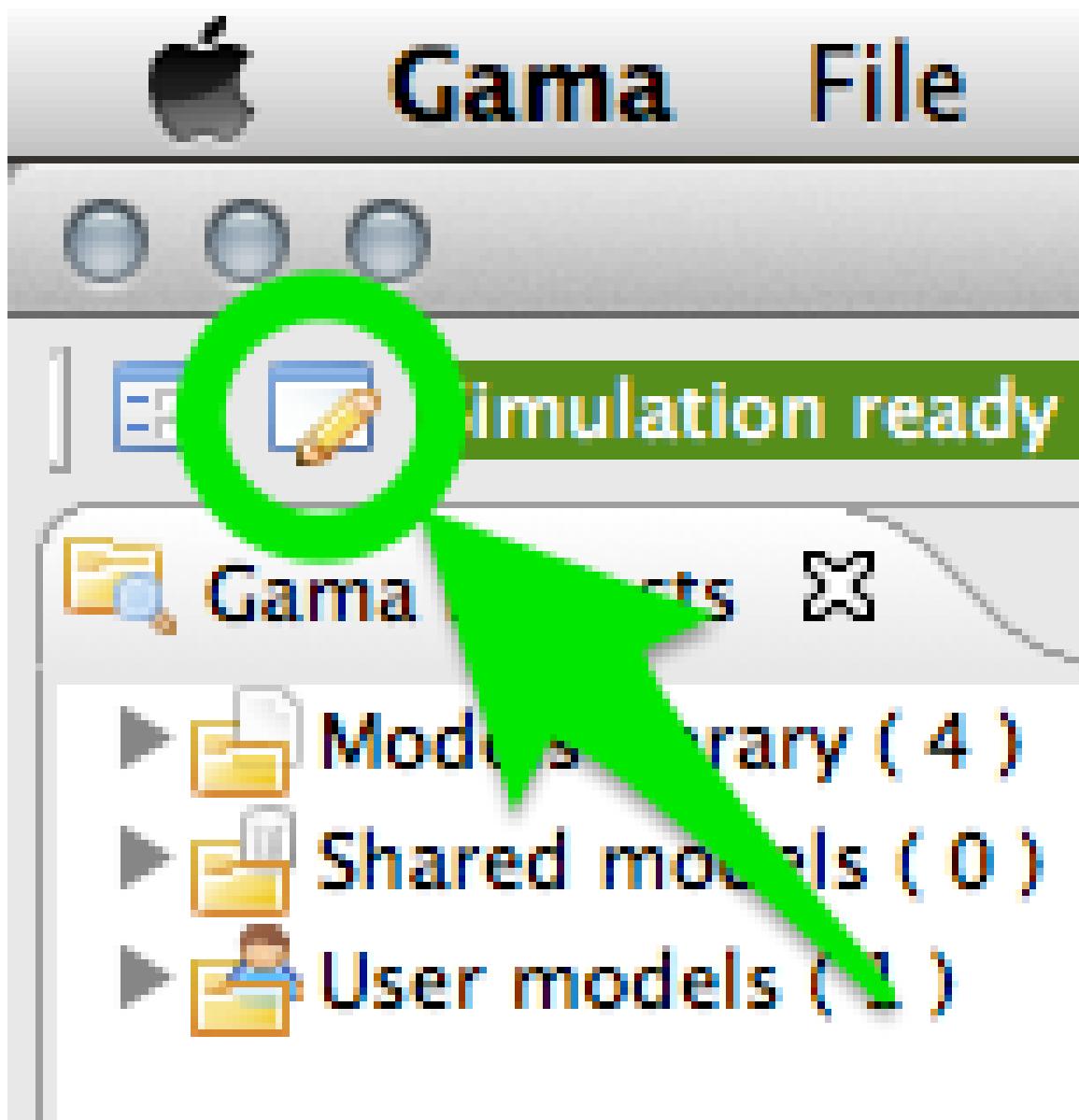
Once an experiment has been launched (unless it is run in [headless mode](#), of course), it normally displays its views and waits from an input from the user, usually a click on the "Run" or "Step" buttons (see [here](#)). It is however possible to make experiments run directly once launched, without 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).



4.2 Experiments User interface

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, next to the [preferences](#) button, 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 [\[G__Display displays\]](#)), 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.

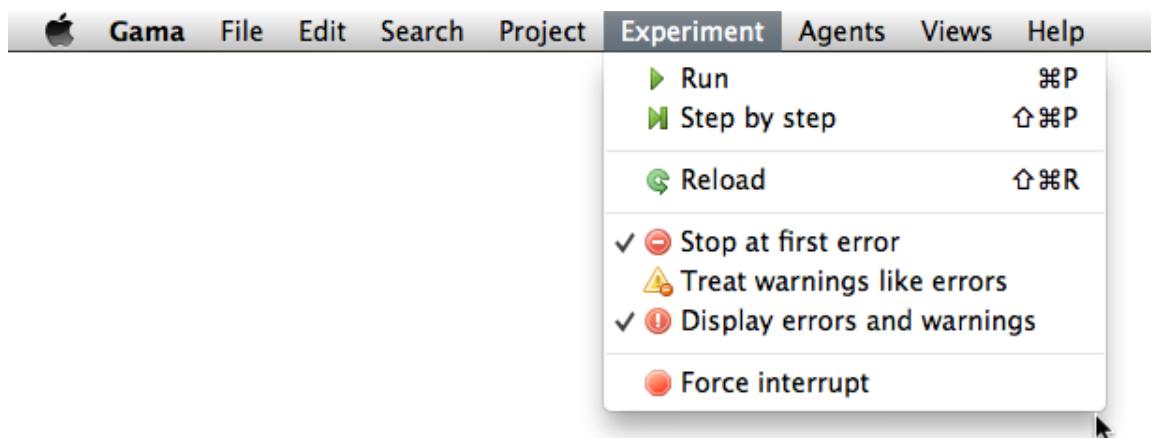
4.2.1 Menus and commands

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.

Experiment Menu

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

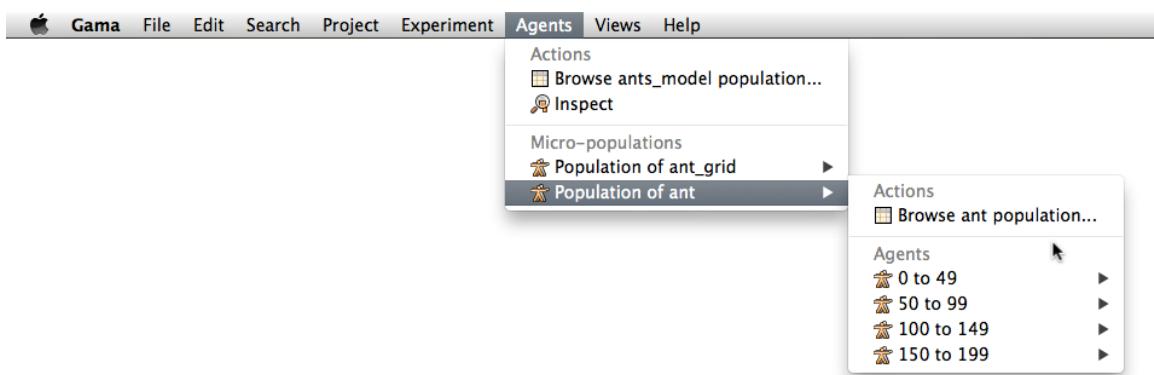


- **Run/Pause** : allows to run or pause the experiment depending on its current state.
- **Step by Step** : runs the experiment for one cycle and pauses it after.
- **Reload** : stops the current experiment, deletes its contents, and reloads it, **taking into account the parameters values that might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

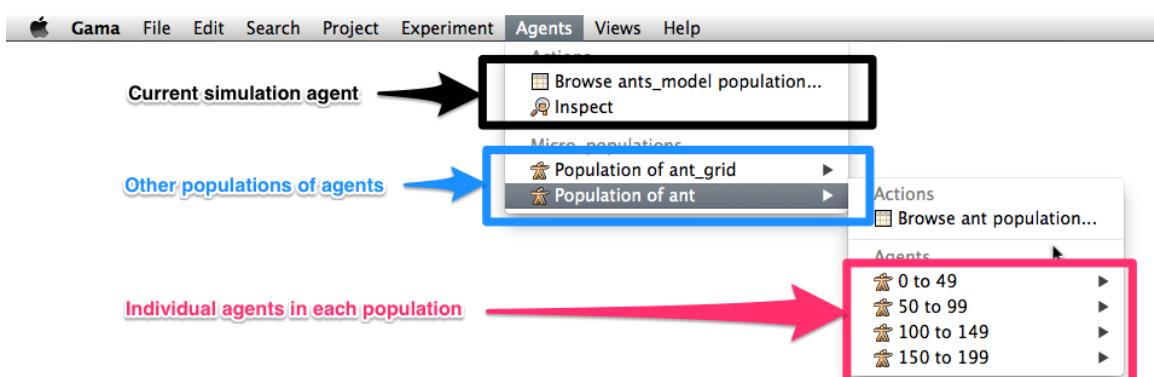
as it can have undesirable effects depending on the state of the experiment (for example, if it is reading files, or outputting data, etc.).

Agents Menu

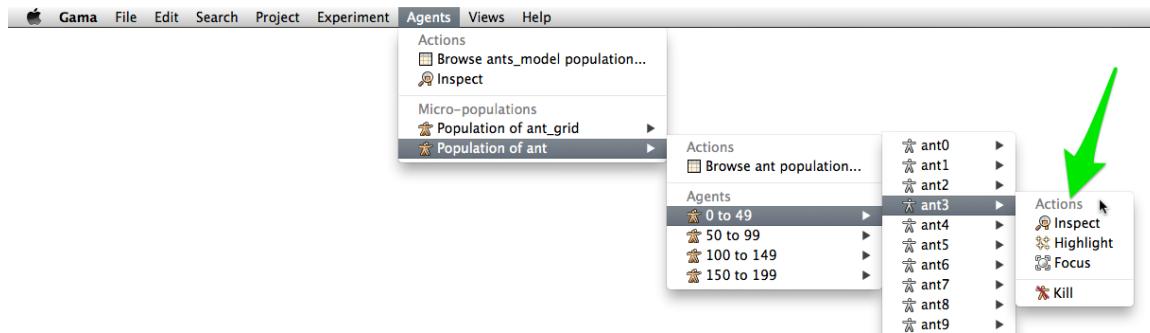
A second menu is added in the simulation perspective: "Agents". This menu allows for an easy access to the different agents that populate an experiment.



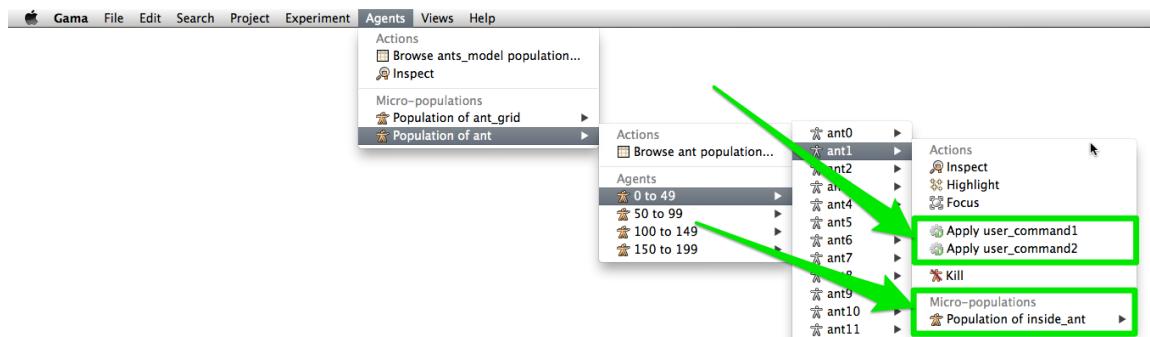
This hierarchical menu is always organized in the same way, whatever the experiment being run. A first level is dedicated to the current simulation agent: it allows to [browse](#) its population or to inspect the simulation agent itself. Browsing the population will give access to the current experiment 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.



On the contrary, if they host other agents (e.g. a population of "inside_ant" in our example) and if their species is provided with user commands, their individual menu will look like the following:

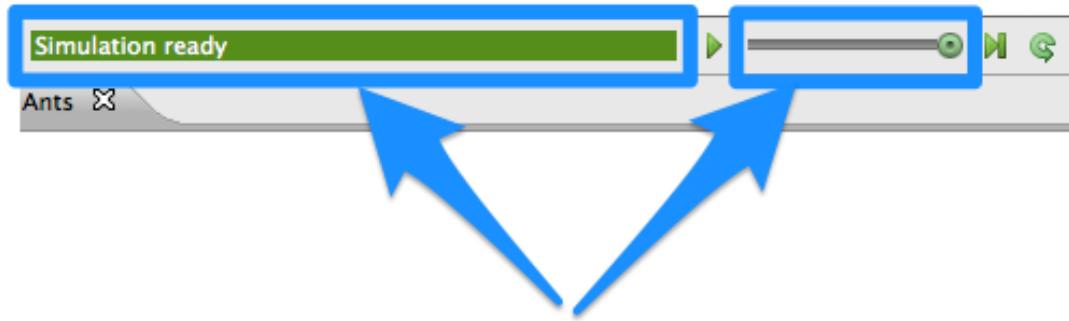


The 4 actions that will be there most of the time, then, are:

- **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.
- **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.

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

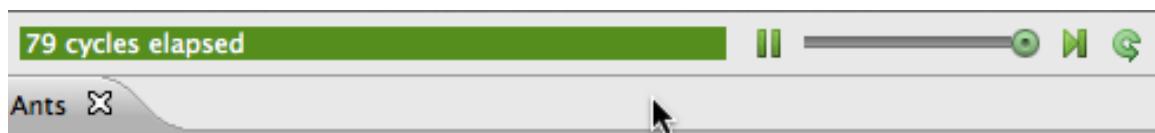


While opening an experiment, the status will display some information about what's going on. For instance, that GAMA is busy instantiating the agents, or opening the displays.

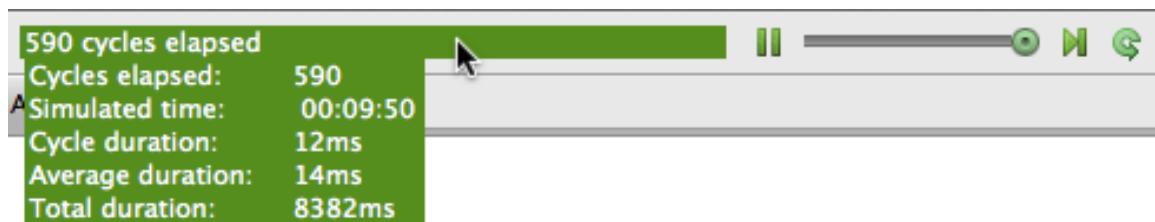
Instantiating agents

Building outputs

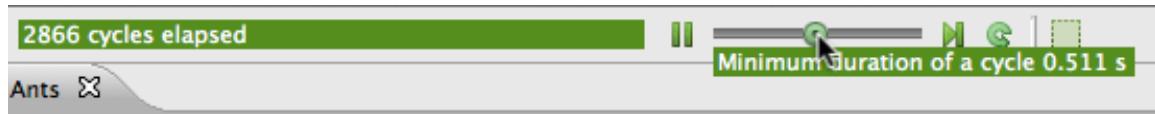
The orange color usually means that, although the experiment is not ready, things are progressing without problems (a red color message is an indication that something went wrong). When the loading of the experiment is finished, GAMA displays the message "Simulation ready" on a green background. If the 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.



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.



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

4.2.2 Parameters view

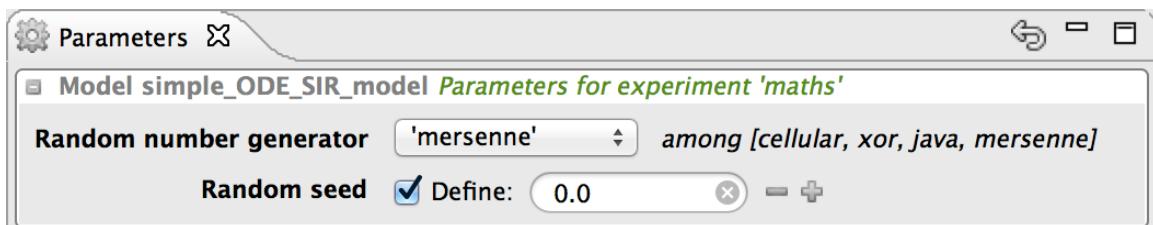
Parameters View

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.**

Built-in parameters

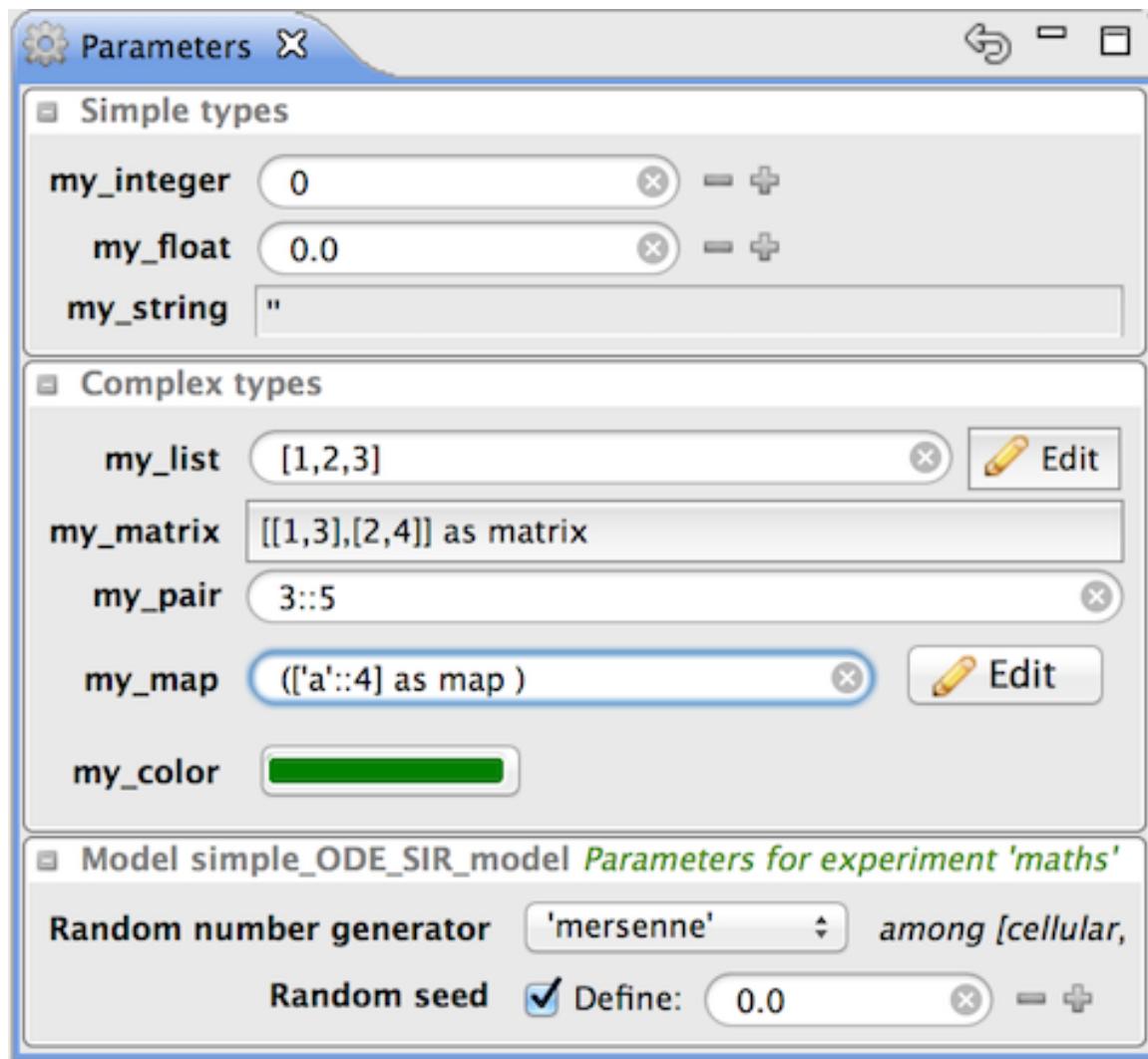
Every [GUI experiment](#) displays a pane named "Parameters" containing at least two built-in parameters related to the random generator:

- the Random Number Generator, with a choice between 4 RNG implementations,
- the Random Seed



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.



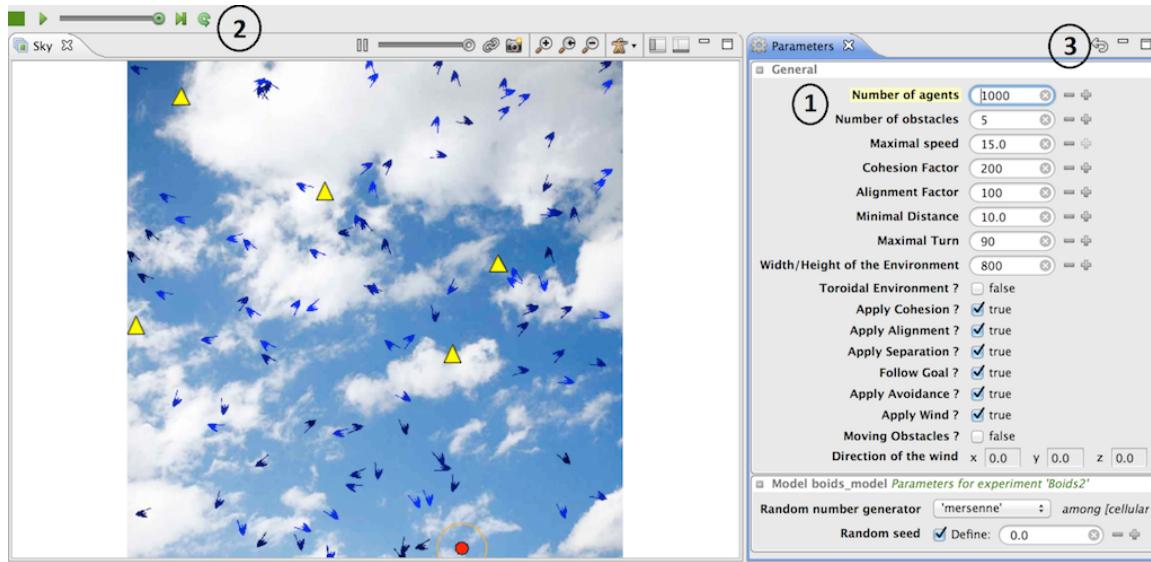
The above parameters view is generated from the following code:

```
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";
    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_map" var: mp <- ["a"]::4 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 parameters value (1 in the image below). In order that the modification to be taken into account in the simulation, he has to reload the simulation (click on the reload button, 2 in the image). If he wants to come back to the initial value of parameters, he can click on the top-right curved arrow of the parameters view (cf. 3 in the image).



4.2.3 Inspectors and monitors

Inspectors and monitors

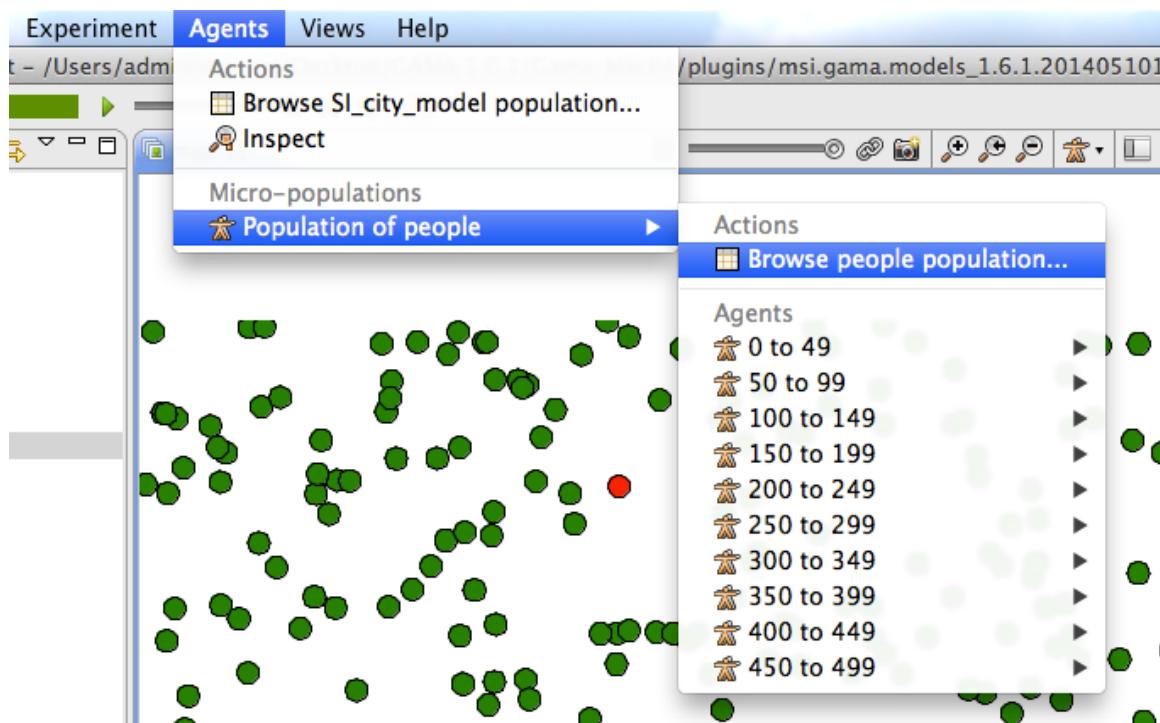
GAMA offers some tools to obtain informations about one or several agents. There are two kinds of tools :

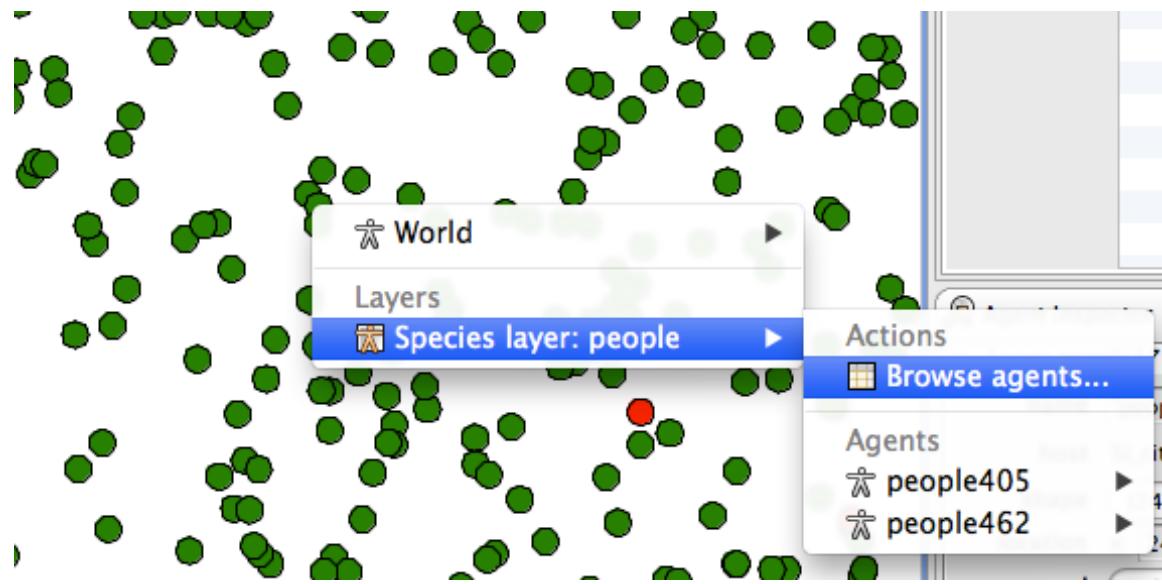
- agent browser
- agent inspector

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

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 by right-clicking on a display.





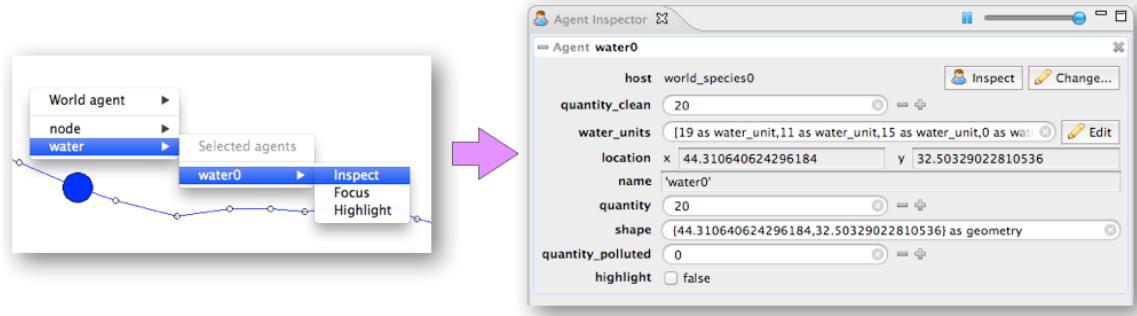
It displays in a table all the values of the agent variables of the considered species; each line corresponding to an agent.

Population of people (500)							
People (500) population in macro-agent SI_city_model0							
people (500)							
Attributes	#	destination	heading	is_infected	location	name	speed
agents	0	{302.38857846...	48	false	{301.45923039...	'people0'	1.3888888888...
destination	1	{69.141959692...	167	false	{70.495251449...	'people1'	1.3888888888...
heading	2	{214.31294721...	278	false	{214.11965124...	'people2'	1.3888888888...
host	3	{157.44714596...	0	false	{156.05825707...	'people3'	1.3888888888...
is_infected	4	{199.49414129...	27	false	{198.25663223...	'people4'	1.3888888888...
location	5	{226.63425053...	120	false	{227.32869498...	'people5'	1.3888888888...
members	6	{70.898468389...	246	false	{71.463380393...	'people6'	1.3888888888...
name	7	{260.94359845...	168	false	{262.30213678...	'people7'	1.3888888888...
peers	8	{132.21554083...	337	false	{130.93706187...	'people8'	1.3888888888...
shape	9	{12.872425192...	187	false	{14.250961514...	'people9'	1.3888888888...
speed	10	{139.98743883...	26	false	{138.73911377...	'people10'	1.3888888888...
	11	{259.98708172...	155	false	{261.24584253...	'people11'	1.3888888888...
	12	{233.19745240...	17	false	{231.86925135...	'people12'	1.3888888888...
	13	{326.81312035...	222	false	{327.84526594...	'people13'	1.3888888888...
	14	{426.93030670...	240	false	{427.62475114...	'people14'	1.3888888888...

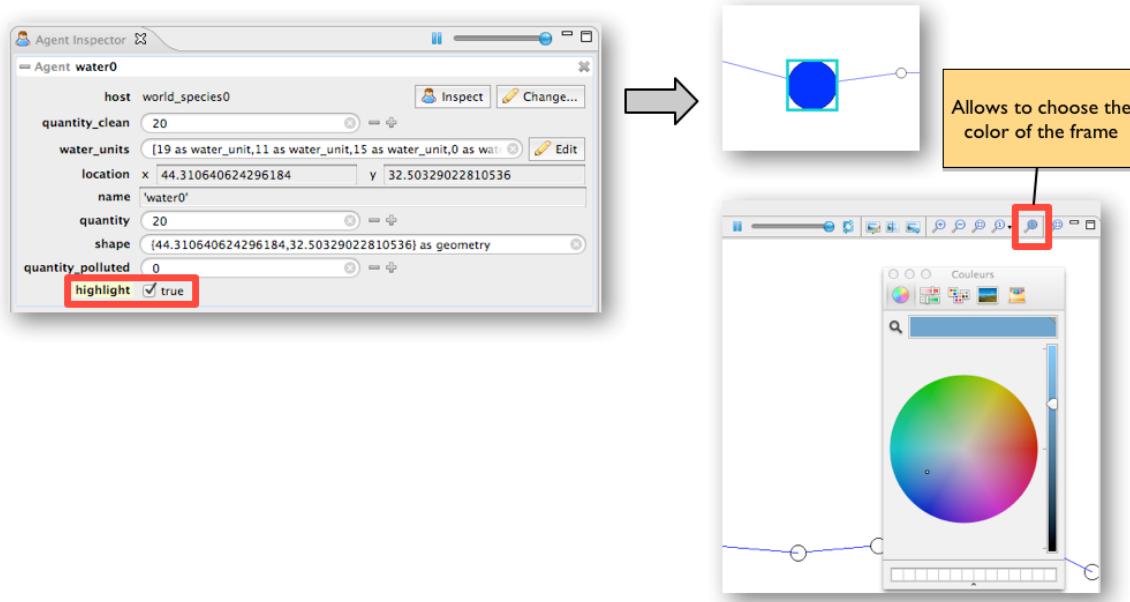
By clicking on the right mouse button on a line, it is possible to highlight or to inspect the corresponding agent.

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.



It is possible to «highlight» the selected agent.

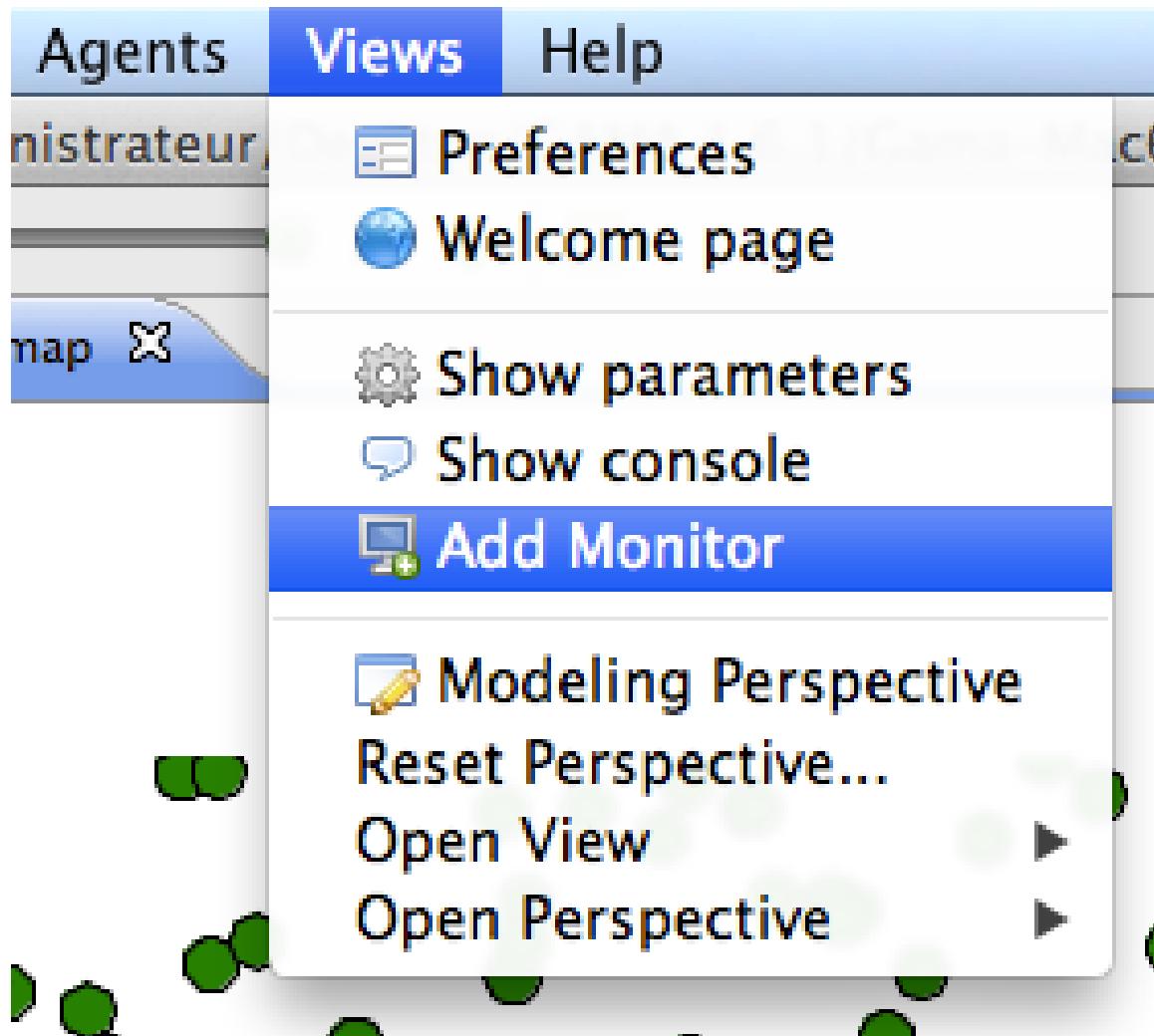


Monitor

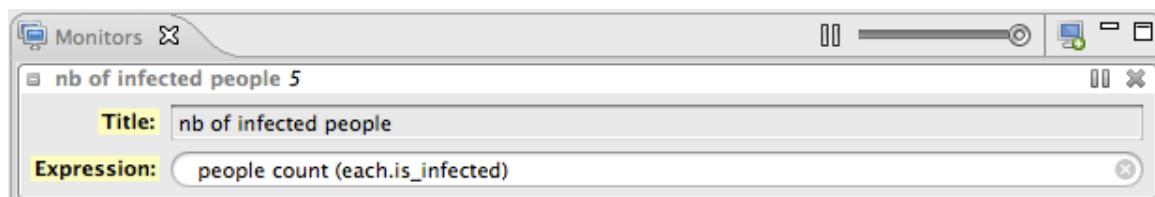
Monitors allow to follow the value of a GAML expression. For instance the following monitor allow to follow the number of infected people agents during the simulation. The monitor is updated at each simulation step.



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. To define a monitor, first choose **Add Monitor** in the **Views** menu, then define the display legend and the expression to monitor.



In the following example, we defined a monitor with the legend "nb of infected people" and that has for value the number of infected people.



The expression should be written with the GAML language. See [this page](#) for more details about the GAML language.

4.2.4 Displays

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.

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:

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

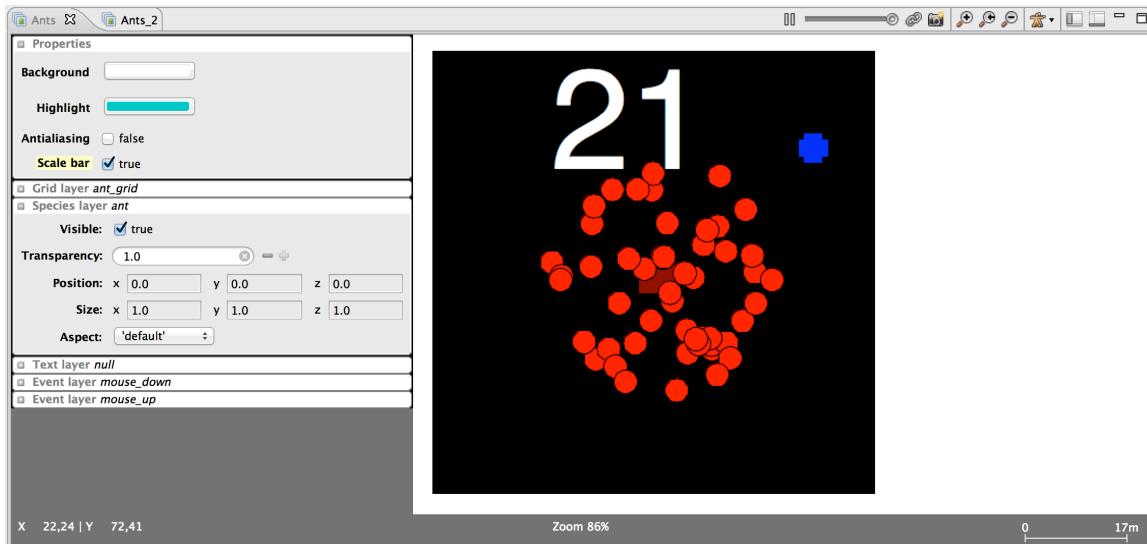
- **Show/hide overlay** .

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:

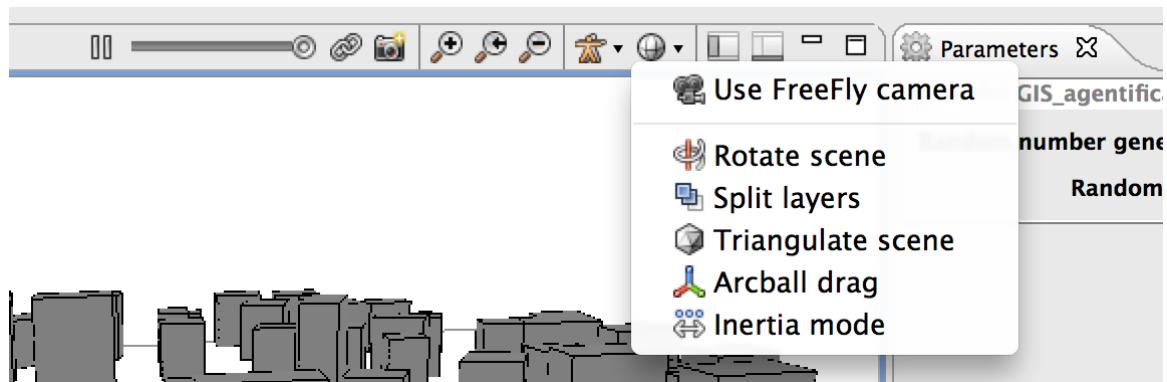
- the position of the mouse in the display,
- the zoom ratio,
- the scale of the display (depending on the zoom).



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,
- **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**
- **Arcball drag** (only with Arcball camera): the right press and hold and mouse move makes rotation on the scene
- **Inertia mode** (only with Arcball camera): in inertia mode, when the modeler stops moving the camera, there is no straight halt but a kind of inertia.



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

4.2.5 Batch Specific UI

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.

Information bar

In batch mode, the top information bar displays 3 distinct information (instead of only the cycle number in the GUI experiment):

- **Run** : 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](#));
- **Simulation** : the number of replications done (and the number of replications specified with the repeat facet);
- **Cycle** : the cycle number in the current simulation.

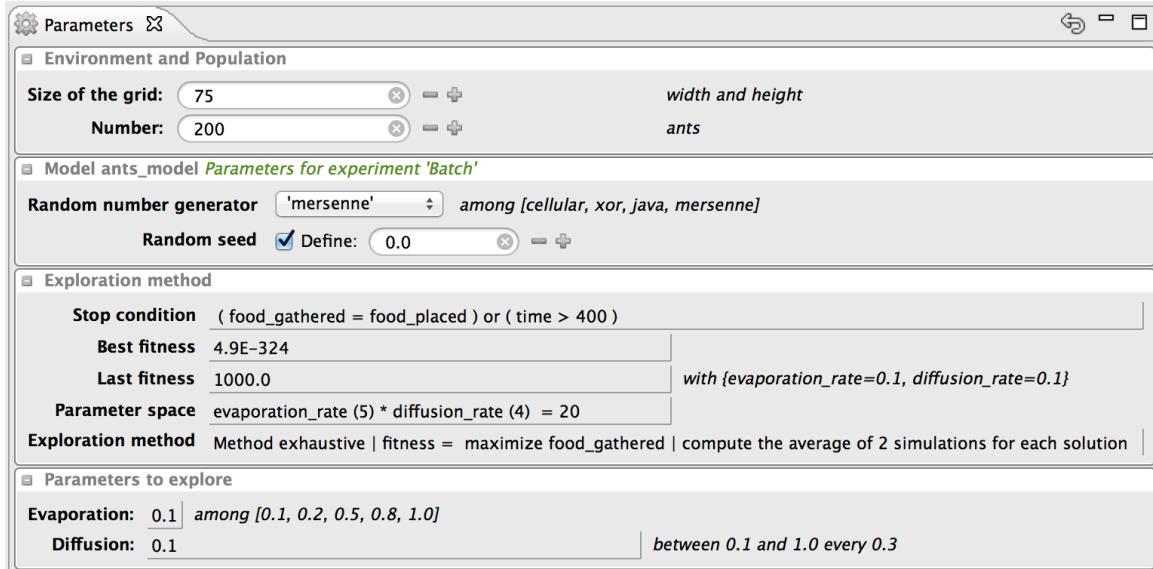


Batch UI

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;
```



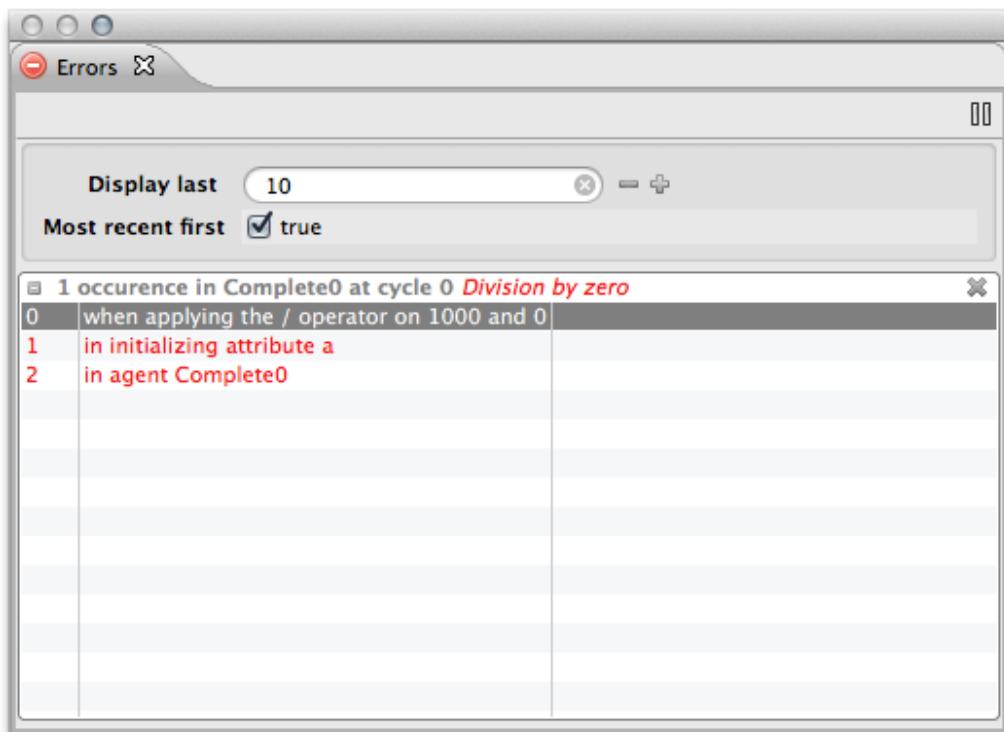
The interface summarizes all model parameters and the parameters given to the exploration method:

- **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).

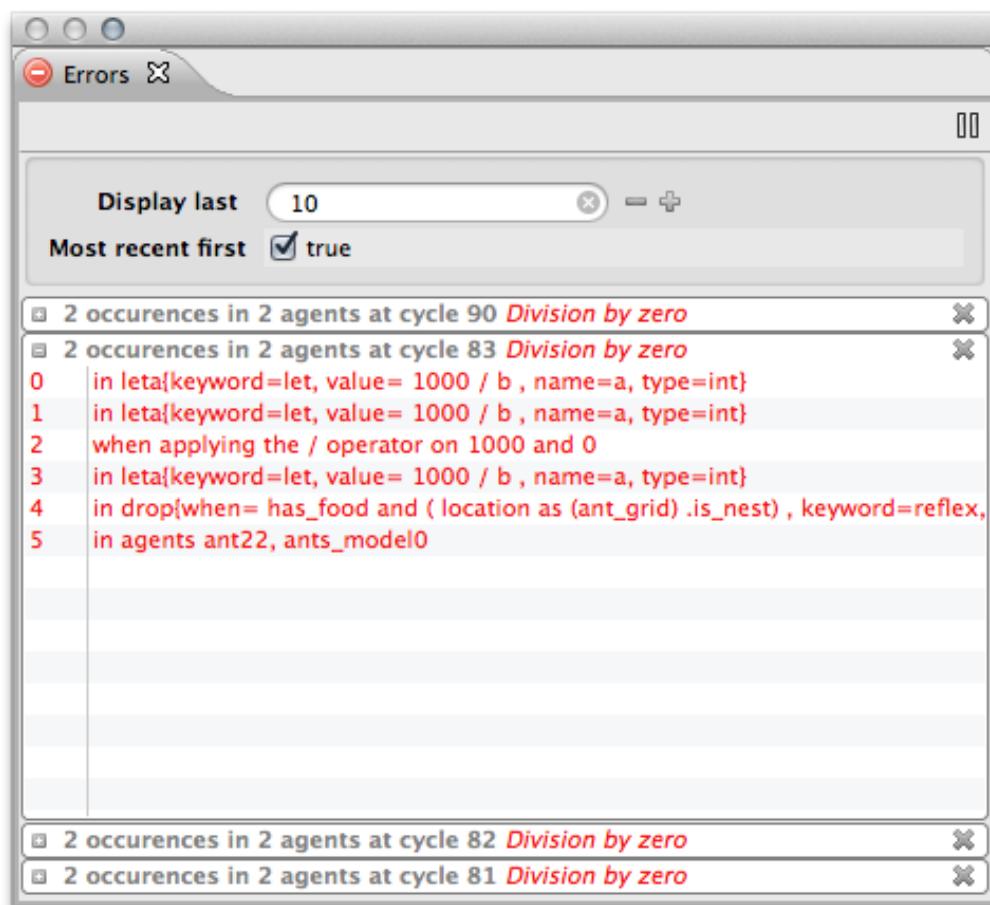
4.2.6 Errors View

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 "statuses" in GAMA, error 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.



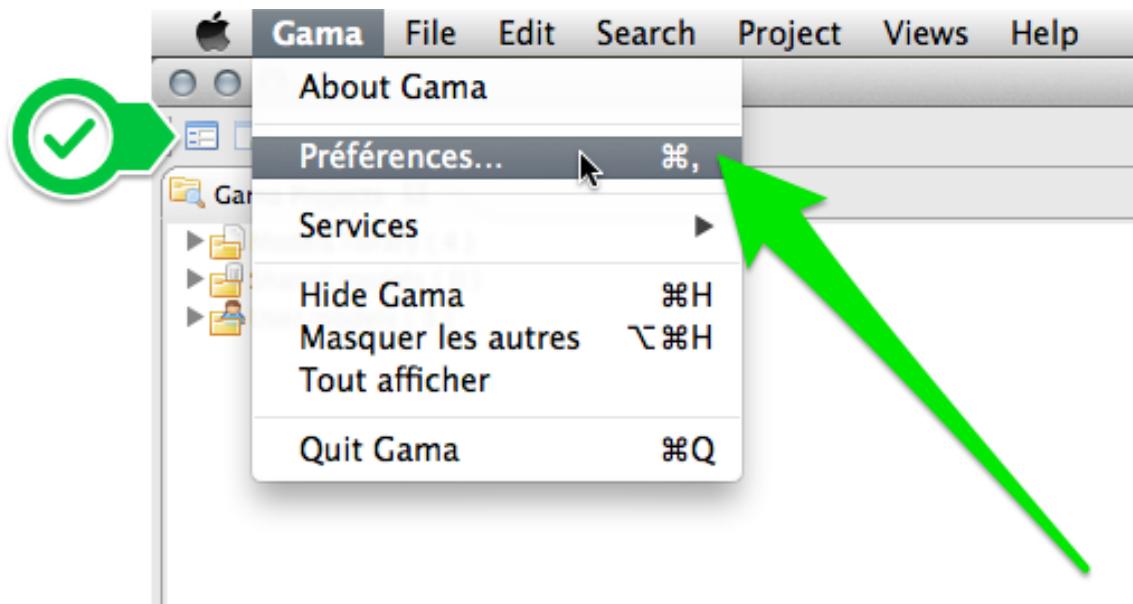
5. Preferences

Preferences

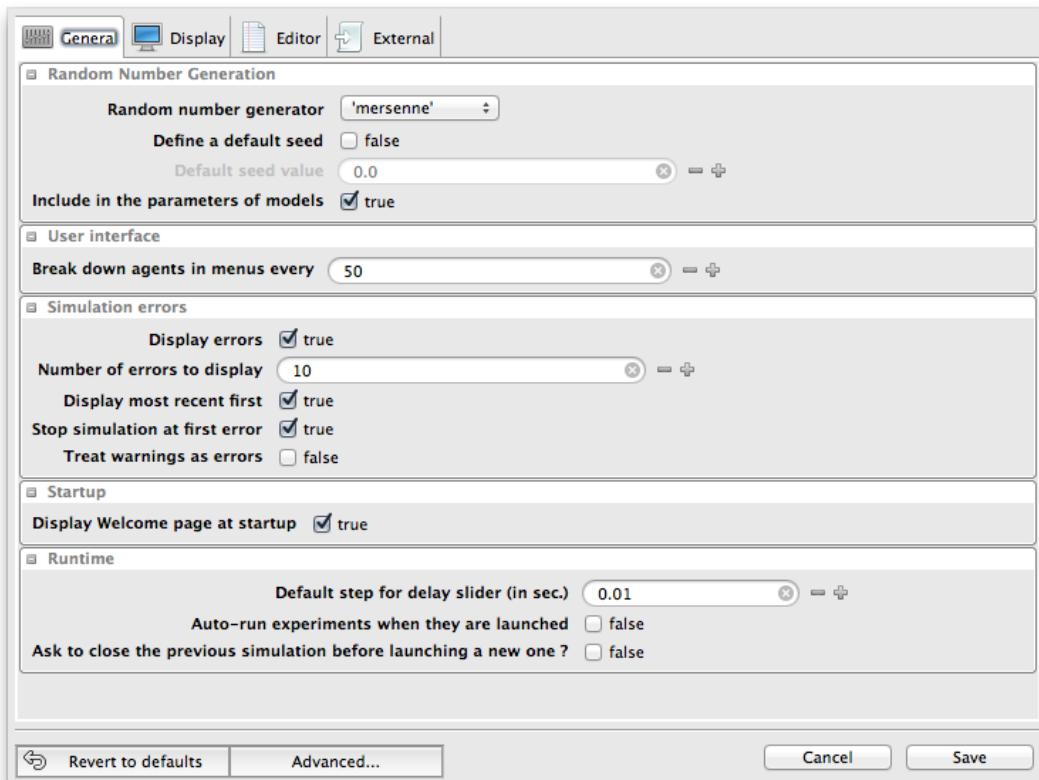
—
Various preferences are accessible in GAMA to allow users and modelers to personalize their working environment. This section review the different preference tabs available in the current version of GAMA, as well as how to access the preferences and settings inherited by GAMA from Eclipse. Please note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

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.



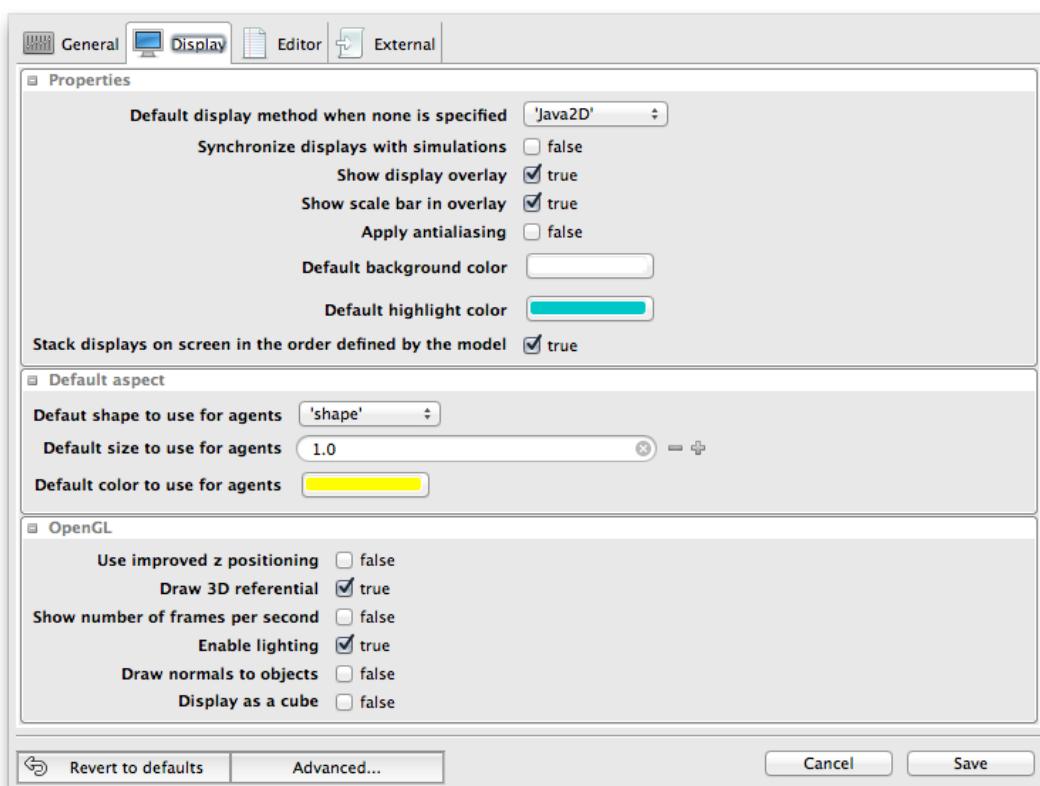
General



- **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.
- **User Interface**
 - 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.
- **Simulation 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).
- **Startup**

- Display welcome page at startup: if true, and if no editors are opened, the [welcome page](#) is displayed when opening GAMA.
 - **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.
-

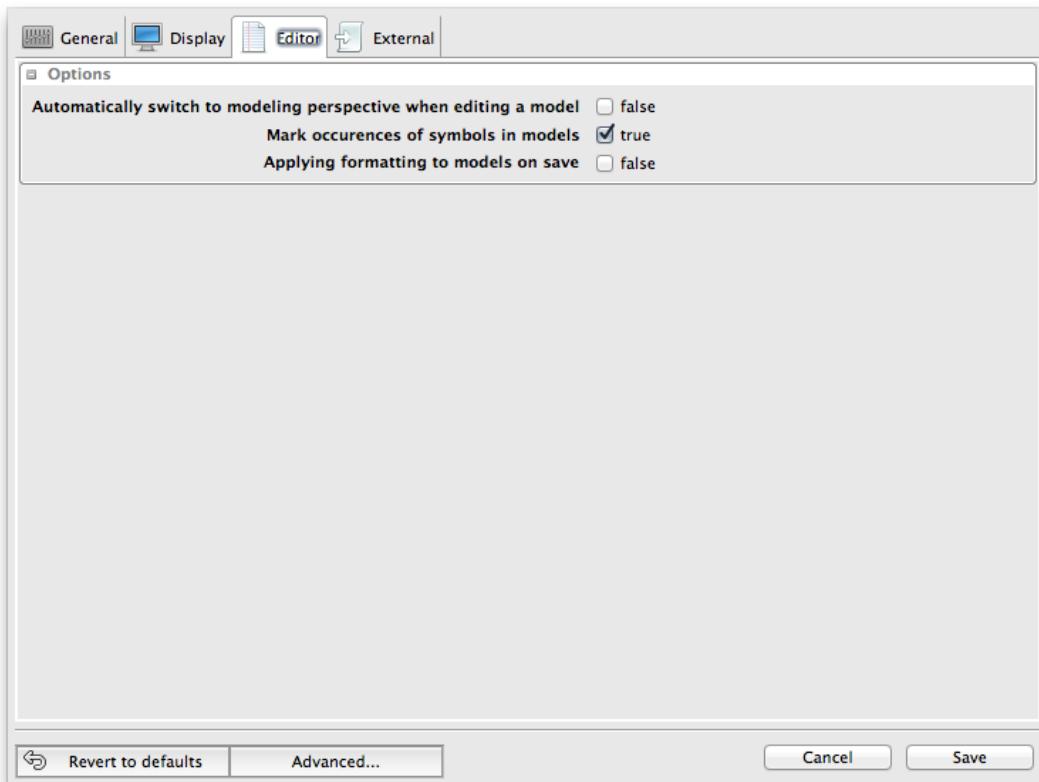
Display



- **Properties** : various properties of displays
 - Default display method: use either 'Java2D' or 'OpenGL' if nothing is specified in the [declaration of a display](#).
 - 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.
- **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.

Editor

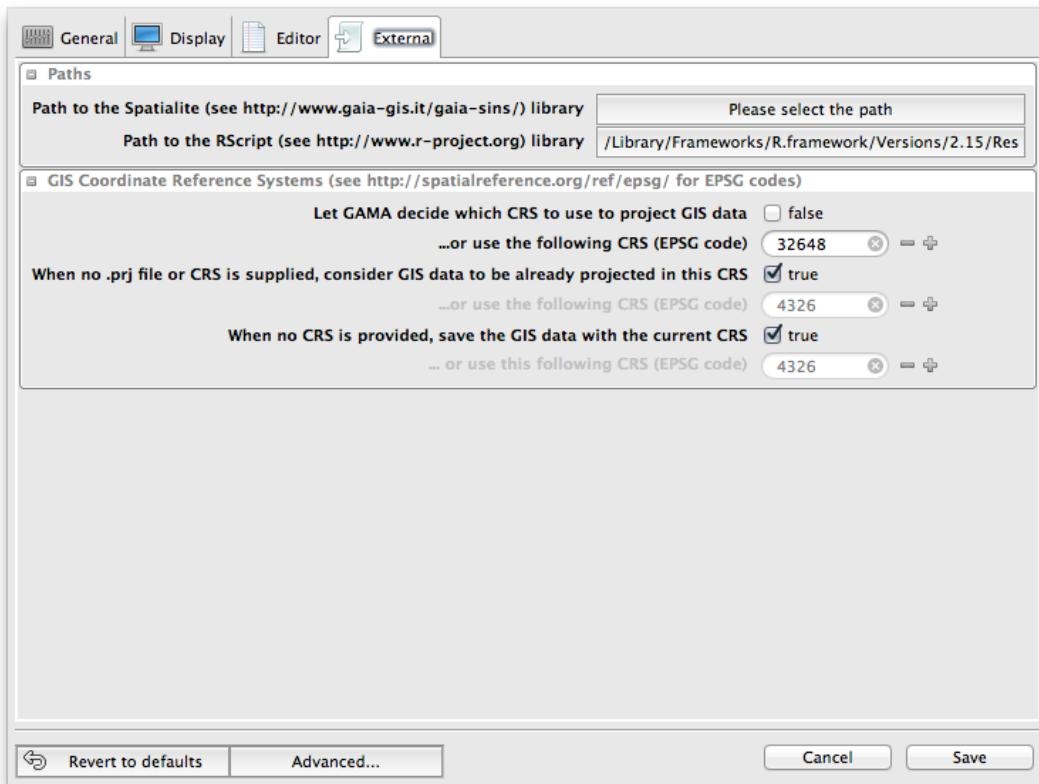


Most of the settings and preferences regarding editors can be found in the [advanced preferences](#).

- **Options**

- Automatically switch to Modeling Perspective: if true, if a model is edited in the Simulation Perspective, then the perspective is automatically switched to Modeling (*inactive for the moment*)
- 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.

External



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

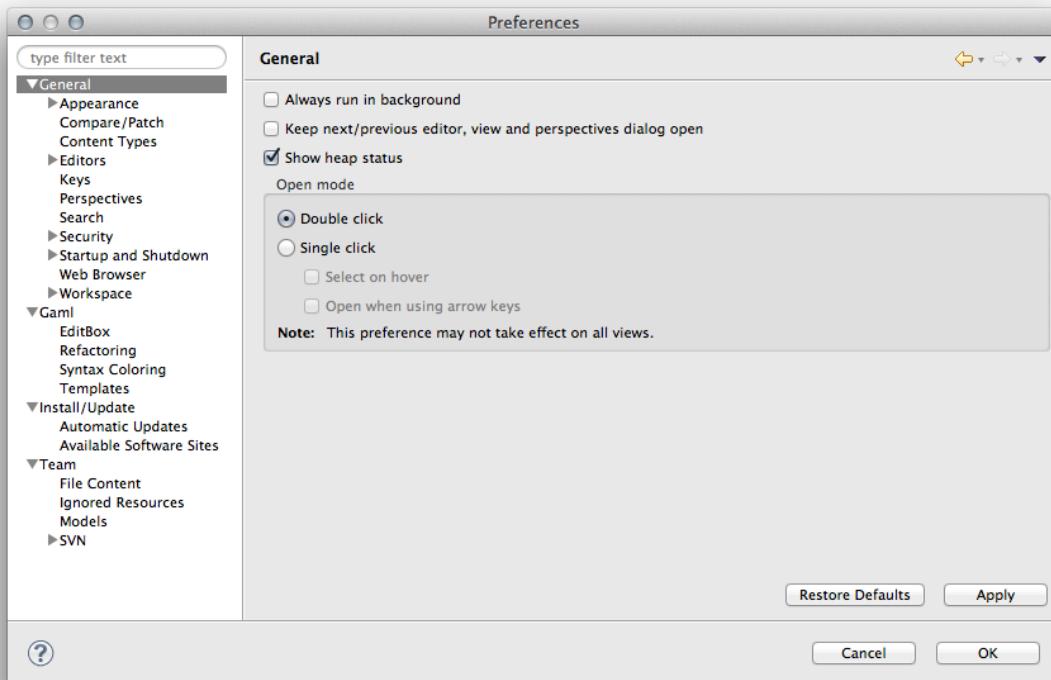
- **Paths**
 - Path to Spatialite: the path to the Spatialite library (<http://www.gaia-gis.it/gaia-sins/>) in the system.
 - Path to RScript: the path to the RScript library (<http://www.r-project.org>) in the system.
- **GIS Coordinate Reference Systems** : settings about CRS to use when loading or 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 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](#)).



GAML (GAMA Modeling Language)

GAML (GAMA Modeling Language)

GAML

Models that users want to simulate in GAMA have to be written in a special language, called **GAML** (short for **GA** ma **M**odeling **L**anguage) GAML is born from the necessity to have a high-level declarative way of defining and reusing structures found in almost all agent-based models. See [here](#) for more information about its background. Although this choice requires users to learn a new programming (or better, *modeling*) language, everything has been made in GAMA to support a short learning curve, so that they can become almost autonomous in a limited time (informal measures taken at the different [G__Events events centered on GAMA] have shown that one day is enough to acquire sufficient skills in writing complete models in GAML). The documentation on GAML is organized in 5 main points:

- Description of the general structure of a model: see [this page](#)
- Description of the declaration of species (and all their components): see [this page](#) and all its subpages
- Description of the declaration of experiments: see [this page](#) for regular experiments and [this one](#) for batch experiments.
- Reference of the [language](#) regarding all the structures provided to modelers
- Recipes of how to use special or advanced features offered in GAML: see [this page](#) .

In addition, some of the fundamental concepts behind GAML are also described in detail, both on the [modeling infrastructure](#) and the [runtime infrastructure](#) on which GAML is relying to run experiments on models.

1. Key Concepts

Key Concepts (Under construction)

—

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.

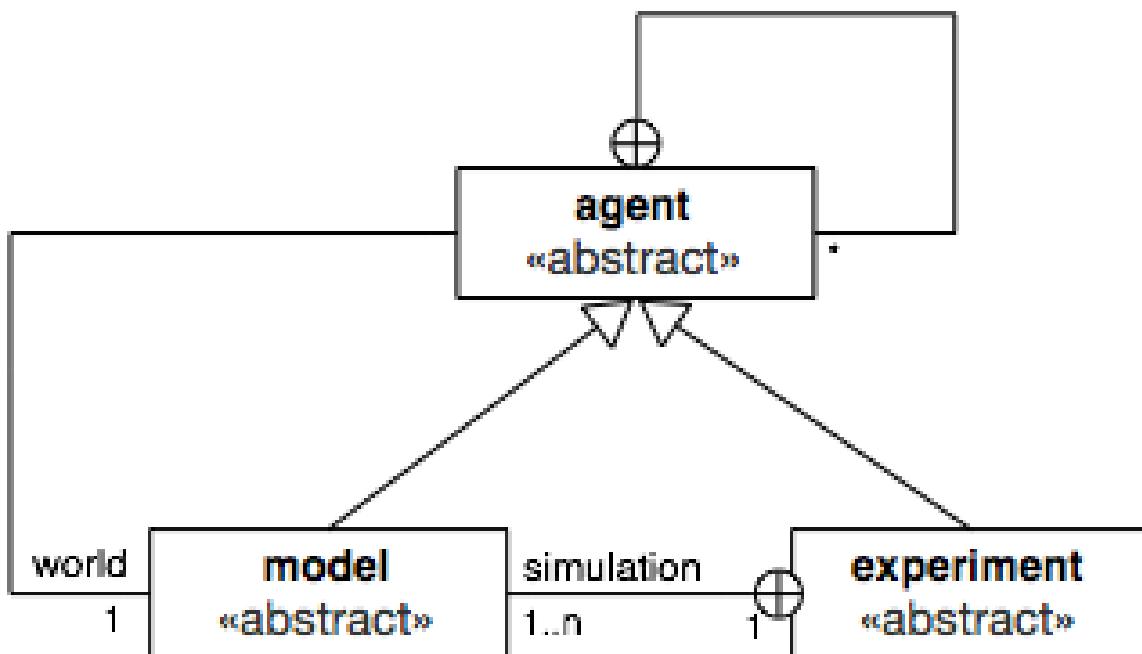
— Lexical semantics of GAML

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

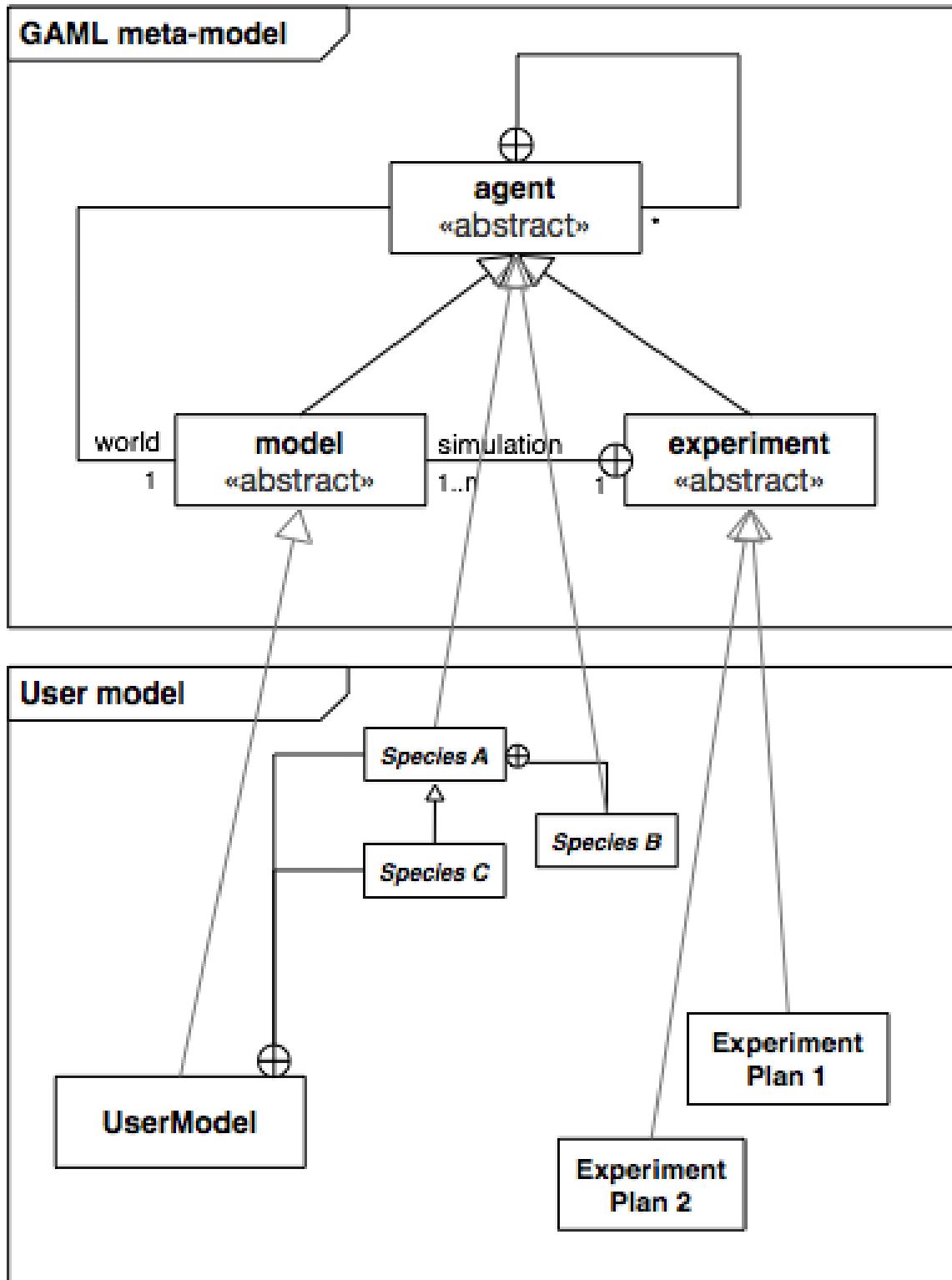
component owning its own data and executing its own behavior, alone or in interaction with other agents).

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:



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].

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
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	-

1.1 Runtime Concepts

Runtime Concepts (Under Construction)

—
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.

Agents Creation

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...
    }
}
```

Scheduling of 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'
```

2. Organization of a Model

Organization of a model (Under construction)

As already extensively detailed in the [key concepts 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

Model Header (_model species_)

The header of a model file begins mandatorily 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, actions and behaviors  
}
```

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. 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, actions, behaviors  
    species A {...}  
    species B {...}  
}
```

—

Experiment declarations

3. Defining Species

Defining Species

Species are key elements of GAMA models (see [here](#)). A species is an archetype of agents and specifies their properties. A model is itself a species that can contain any number of species.

Species Declaration

The simplest way to declare a regular species is the following:

```
species a_name {  
}
```

for example:

```
species foo {} //it is also possible to directly write: species foo;
```

The agents that will belong to this species will only be provided with some built-in attributes and actions, a basic behavioral structure and nothing more. These elements are directly inherited from the default parent species called '[agent](#)' . See [[G__RegularSpecies](#) this page] to specify a different parent for the species. A species can contain several elements:

- [Attributes](#) : define the state of the agents.
- [Actions](#) : define a capability of the agents. An action is a block of instructions that are executed when the action is called.
- [Inits and Reflexes](#) : defines the default behavior of the agents. Both statements contain instructions that are executed, respectively, once when the agent is created for init statements, and at each simulation step (according to an optional condition) for reflex statements.
- [Aspects](#) : define how the agents can be displayed.
- [Equations](#) : define a set of differential equations that can be integrated when necessary.
- [Micro-species](#) : nested species can be described inside a species. See [[G__MultiLevelArchitecture](#) here] for more details on the relationships between macro- and micro-species.

Note that all the elements previously defined are optional. It is totally possible to define an empty species without any attributes, actions, reflexes, aspects, equations or micro-species like in the example above. In addition to the regular inheritance mechanisms, modelers can attach skills and control architectures to species, which will provide their agents with new attributes, actions and behaviors. See [here](#) for how to define these capabilities. Finally, some specific features concerning

the interaction of users with agents can be added to any species. It is for example possible to define an action that will be executed by the user. See [here](#) for more details.

Scheduling Description

The modeler can specify the scheduling information of a species. This scheduling information is composed of the execution frequency and the list of agent to be scheduled.

- the execution frequency is the frequency which agents of the species are considered to be scheduled.
- "the list of agent to be scheduled" is an expression returning a list of agent dynamically evaluated at runtime.

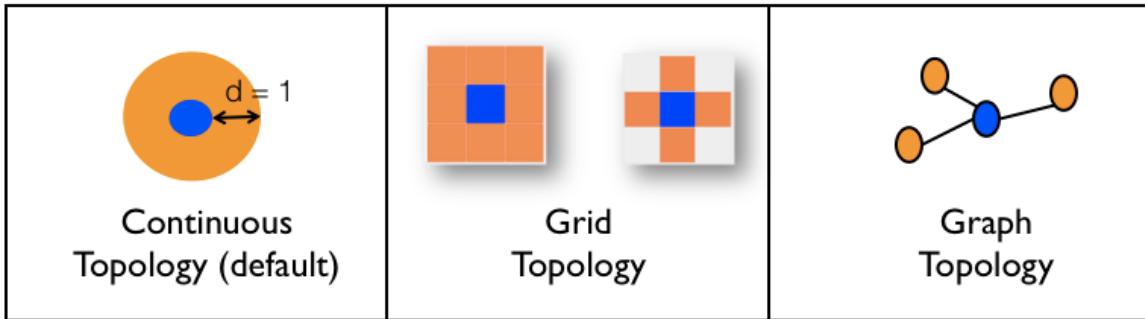
```
species foo frequency: 2 schedules: foo where (each.energy > 50) {  
    float energy <- rnd (100) min: 0 max: 100 value: energy - 0.001;  
    ...  
}
```

- frequency: we schedule the agents declared in schedules: every 2 simulation steps (or all the agents of the species if schedules: is not defined).
- schedules: an expression that returns the list of agent to be scheduled, in that case all the foo agents having a level of energy greater than 50. Two things worth to be mentioned regarding this facet:
 1. The list of agents can contain any type of agents, not necessarily agents of the species, making it possible to define [custom scheduling rules](#).
 2. The contents of this facet **is not inherited** by children species.

Topology Description

The topology describes the spatial organization of the species. This imposes constraint on the movement and perception (neighborhood) of the species' agents. GAMA supports three types of topology: continuous, grid and graph.

Distance = 1



```
species foo topology: (square (10)) at_location {50, 50} {
    ...
}
```

Topology of the "foo" species is a square of 10 meters each side at location {50, 50}. By default, a species has a continuous topology. However, it is possible to define species with a specific grid or graph topology (see the next Section).

Types of Species

Several types of species exist in GAMA:

- **global species** : The global species defines the model, i.e. the attributes, actions, behaviors and micro-species that describe the world agent.
- [G__RegularSpecies regular species] : species of agents with a continuous topology by default.
- **grid species** : species of agents with a grid topology.
- **graph species** : species of agents with a graph topology.
- **Mirror species** : species of agents that mirror the population of another species

3.1 Global Species

Global Species (Under Construction)

```
<font color="red"> todo :  
  
• move all the attributes to the 'model'/'experiment' species descriptions  
• move all the actions to the 'model' species descriptions  
• proofread the language (sometimes terrible to read !)  
  
</font>
```

—
The global species defines 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 run an experiment and that initializes the simulation through its **init** section. 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.

Declaration

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

```
global {  
}
```

Like for other species, user can define several element inside the global section:

```
species a_name {  
    [attribute declarations]  
    [init]  
    [action declarations]  
    [behaviors]  
    [aspects]  
}
```

In the same way, user can through facets attach skills, a control architecture, define scheduling rules like for other species. See [this page](#) for more details. A facet that is specific to the global species is the **torus: true/false** one that defines the toroidal properties of the environment. By default (if this facet is not used) the environment is not torus. Example:

```
global torus: true {
}
```

Here an example of a global section:

```
global {
    float number_of_bugs <- 1000;
    init {
        create bug number: number_of_bugs ;
    }
}
```

In this case, the world agent will have only one attribute and will, when the user will run an experiment, create **number_of_bugs** bug agents.

Environment Size

The attribute shape of the global species allows to define the global environment. GAMA supports three types of topologies for environments: continuous, grid and graph. By default, the world agent has a continuous topology and its geometry is a rectangle of size 100mx100m. The geometry of the environment can be defined:

- using a geometry: geometry shape <- rectangle(300,100);
- using a shapefile or an OSM file (GIS): envelope of all the data contained in the GIS file: geometry shape <- envelope("bounds.shp"); ,
- using a raster file (asc): geometry shape <- envelope("bounds.asc"); ,

Example:

```
global {
    geometry shape <- circle(50);
}
```

```
global {
    file road_shapefile <- file("../includes/roads.shp");
    geometry shape <- envelope(road_shapefile);
}
```

```
global {
    file mnt_file <- file("../includes/mnt.asc");
    geometry shape <- envelope(mnt_file);
}
```

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.

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.

```
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)};  
...  
}
```

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;
    }
}
```

Scheduling

In terms of scheduling the world agent is activated first at each time step. It means that the world first acts (update of its attributes, execution of its reflexes), then the agents of the other species act (in random order by default). It is the same for the init section: first the init section of the world agent is executed, then if agents are created during this execution, the init section of these agents are executed.

3.3 Grid Species

Grid Species

```
<font color="red"> todo :
  • constraints: grid are only micro-species of the world
  • link with matrix
  • maybe a word about scheduling ?
</font>
```

—
A grid is a particular species of agents. Indeed, a grid is a set of agents that share a grid topology. As other agents, a grid species can have [attributes](#), [\[G__DefiningActions actions , behaviors , aspects , equations\]](#) and [micro-species](#). 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 is provided with a set of additional built-in variables

Declaration

basis

There are several ways to declare a grid species:

- by defining the number of rows and columns:

```
grid grid_name width: nb_cols height: nb_rows neighbors: 4/6/8 {
    ...
}
```

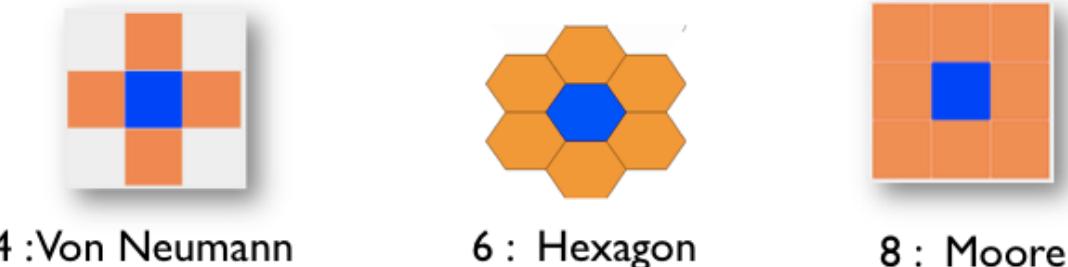
- by using an `asc` grid file:

```
grid file:file("../includes/dem.asc") neighbors: 4/6/8 {
    ...
}
```

With:

- `width` : number of cells along x-axis (number of columns)

- height : number of cells along y-axis (number of rows)
- file: file used to initialized the grid
- neighbours : neighborhood type (4 - Von Neumann, 6 - hexagon or 8 - Moore)



If a file is used to declare the grid, the number of rows and columns is automatically computed from the file. In the same way, the value of the cell read in the file is stored in the **grid_value** built-in variable.

optimization facets

A grid can be provided with specific facets that allows to optimized the computation time and the memory space.

use_regular_agents

```
use_regular_agents: true/false,
```

If **false**, the grid will be composed of "minimal" agents, that uses less computer memory, but have some limitations: they cannot inherit from "normal" species, they cannot have sub-populations, their name is fixed and cannot be changed,... By default, this facet is set to **true** when declaring a grid without this facet.

use_individual_shapes

```
use_individual_shapes: true/false,
```

If **false**, then just one geometry is used for all agents (that can be translated). The goal is limit the memory used by the geometries. By default, this facet is set to **true** when declaring a grid without this facet.

use_neighbours_cache

```
use_neighbours_cache: true/false,
```

If **false**, no neighbours computation result are stored in the cache. By default, this facet is set to **true** when declaring a grid without this facet.

Built-in variables

Grid agents are provided with several specific built-in variables.

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:

```
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") ;
        }
    }
}
```

```
}
```

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 cells

there are several ways to access to a specific cell:

- by a location: by casting a location to a cell

```
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 [20,10] :" + cell[20,20];
    }
}
grid cell width: 10 height: 10 neighbors: 4 {
```

3.4 Graph Species

Graph Species (Under Construction)

 todo :

- make references to the topology and maybe better explain the purpose of 'graph' species: build a graph using agents (and not — only — between agents).
- link with the two built-in species mentioned
- give constraints (no other topology can be defined)

Using a **graph** species enables to easily shows interaction between agent 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 agent. To instantiate this **graph** species, several step 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** . This method returns true or false according to a given condition that will express the distance between two agents. This distance can be of course the euclidean distance (but in this case it is recommended to use the operator [as_distance_graph](#) but also any other distance define by the user.

Declaration

```
species A parent: graph_node edge_species: edge_agent{
    bool related_to(A other){
        using topology(world) {
            return (self.location distance_to other.location) < 10;
        }
    }
}
```

Define the species used to represent the edges of the generated graph

```
species edge_agent parent: base_edge {
    aspect base {
        draw shape color: rgb("green");
    }
}
```

Example

```
model GraphSpecies
global {
    init{
        create A number:100;
    }
}
species A parent: graph_node edge_species: edge_agent{
    bool related_to(A other){
        using topology(world){
            return (self.location distance_to other.location) < 10;
        }
    }
    aspect base {
        draw sphere(1) color: #blue;
    }
}
species edge_agent parent: base_edge {
    rgb color;
    aspect base {
        draw shape color: rgb("green");
    }
}
experiment graphExp type: gui {
    output {
        display graphView type: opengl{
            species A aspect: base;
            species edge_agent aspect:base;
        }
    }
}
```

3.5 Mirror Species

Mirror Species (Under Construction)

 todo :

- explain the population management a little bit more in details
- explain the constraints (no mirror of itself, no 'create', no reciprocal mirrors — otherwise problems)

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.

Declaration

Given any already declared species

```
species A {  
...  
}
```

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

```
point location <- target.location update: target.location;
```

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

This model simply instantiates 100 agents of the species A that has a moving skills and are wandering at each step. On top of this species a mirror species B is created.

```
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{
    point location <- target.location update: target.location;
    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;
        }
    }
}
```

3.6 Multi-level Architecture

Multi-level Architecture (Under Construction)

 todo :

- Explain a little bit more what the meaning of the "multi-level" architecture is.
- Link to the general architecture of GAMA (the "world" agent being the "host" of regular agents, etc.)
- Perhaps choose a simple example (with no "birds", etc.) but only micro_agent and macro_agent
- explains what it means in terms of scheduling and topology

—
The multi-level architecture offers the modeller the following possibilities : (1) the declaration of a species as a micro-species of another species, (2) the representation of an entity as different types of agent (i.e., GAML species), (3) the dynamic migration of agents between populations.

Details

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 group {
    ...
    species bird_in_group {
        ...
    }
    ...
}
```

In the above example, "ball_in_group" is a micro-species of "group" species. An agent of "group" species can have agents "ball_in_group" species as micro-agents. Agents of "ball_in_group" species have an agent of "group" species as "host" agent.

Representation of an entity as different types of agent

Let's imagine two scenarios of a "bird" entity. Firstly, when a "bird" entity flies alone the modeler would like to represent it in one way (e.g., with a lot of detail). Secondly, when a "bird" entities flies in group (a group of birds is in fact a set of nearby flying bird), the modeler would like to represent it in another way (e.g., less detail and introduce some constraints between nearby birds). In GAML, the modeler can program this as follows:

```
species bird {
    ...
}

species group {
    ...

    species bird_in_group parent: bird {
        ...
    }
    ...
}
```

The "bird" entity is represented as two species: "bird" and "bird_in_group". The special point to remember here is that "bird_in_group" must be a sub-species of "bird" species.

Dynamic migration of agents

In our example, a "group" entity is composed of nearby flying "bird" entities. When a "bird" entity approaches a "group" entity, this "bird" entity will become a member of the group. To represent this, the modeler lets the "bird" agent change its species to "bird_in_group" species. The "bird" agent hence becomes a "bird_in_group" agent. To change species of agent, we can use one of the following statements : [capture](#) , [release](#) , [migrate](#) . For example, in this case, to model the fact that a "group" agent captures the nearby "bird" agents (within a 5 meters distance from the group) as "bird_in_group" agents. We can write something as follows:

```
species bird {
    ...
}

species group {
    ...

    species bird_in_group parent: bird {
        ...
    }

    reflex capture_nearby_free_birds {
        list<bird> nearby_birds <- (bird overlapping (shape + 5));
        if !(empty (nearby_birds)) {
            ...
        }
    }
}
```

```

        capture nearby_birds as: bird_in_group;
    }
}
...
}
```

Access to micro-agents, host agent

An agent can access to its micro-agents and host thanks to two built-in variables : "members" and "host". For example, a "group" agent can access to its micro-agents of "bird_in_group" species as follows :

```

species group {
    ...

    species bird_in_group parent: bird {
        ...
    }

    species other_micro_species {
        ...
    }

    reflex print {
        let<bird_in_group> my_bird_members <- members of_species
bird_in_group;
        write my_bird_members;
    }
    ...
}
```

A "bird_in_group" agent can access to its host as follows:

```

species group {
    ...

    species bird_in_group parent: bird {
        reflex print_host {
            write 'my host agent is ' + host;
        }
    }
    ...
}
```

3.7 Attaching Skills and Control

Attaching Skills and Control

GAMA allows to attach skills and a control architecture to agents through the facets **skills** and **control**. Skills are built-in modules 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. Control are agent control architectures that can be used in addition to the [common behavior structure](#).

Skills

A declaration of skill is done by filling the skills facet 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. The list of available skills in GAMA is:

- moving: for agents that need to move.

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 (r/o)" 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;
}
```

A complete description of existing skills is available [here](#).

—

Control Architecture

GAMA integrates several agent control architectures that can be used in addition to the common behavior structure:

- weighted_tasks
- sorted_tasks
- probabilistic_tasks
- fsm
- user_only
- user_first
- user_last

The choice of an architecture (that is optional) is made through the **control** facet:

```
species ant control: fsm {
    ...
}
```

A description of all existing control architectures is available [here](#).

3.8 Defining Attributes

Defining Attributes (Under Construction)

 todo :

- move temp variables elsewhere
- rewrite "Reserved Keywords" (as most of the rules are now obsolete)
- remove the 'type' subsection as it is now obsolete as well
- link to data types for the default values
- add a paragraph on systematic casting (with examples)

—
Attributes define the internal state of the agents of the species.

Attribute Declaration

An attribute is declared using the following syntax:

```
datatype var_name [optional_facets: ...];
```

In this declaration, datatype refers to the name of a built-in type or a species declared in the model. The value of var_name can be any combination of letters and digits (plus the underscore, " _ ") that does not begin with a digit and that follows certain rules (see "Naming variables"). Examples of valid declarations are:

```
int i;
list my_list;
my_species_name an_agent_of_my_species; // if my_species is declared in the
model as a species.
```

These attributes are given default values at their creation, depending on their datatype: Default Value

int	float	bool	string	list	matrix	point	rgb	graph	geometry
0	0.0	false	"	[]	nil	nil	black	nil	nil

init or <-

When it is necessary to initialize the attribute with another value than its default value, the init (or <-) facet can be used.

```
datatype var_name <- initial_expression [optional_attributes:...];
```

The initial_expression is expected to be of the same type as the attribute (otherwise it is casted to the datatype). Its only (obvious) restriction is that it cannot refer to the attribute being declared. Examples of valid declarations are:

```
int i <- 0;
list my_list <- [i + 1, i + 2, i + 3];
agent an_agent <- self;
```

const

If the value of the attribute is not intended to change over time, it is preferable to declare it as a constant in order to avoid any surprise (and to optimize, a little bit, the compiler's work). This is done with the const facet:

```
const var_name type: datatype <- initial_expression
[optional_attributes:...];
```

With this declaration, the attribute var_name will keep the result of initial_expression as its value for the whole execution of the simulation.

update

What if, on the contrary, the value of the attribute is supposed to change over time and the modeler wants to define this evolution? The update facet is precisely available for this purpose. Basically, its contents is evaluated every time step and assigned to the variable. It means that, unless the contents of this attribute refers to the attribute itself, every modification made in the model to the value of the attribute will be lost and replaced with the evaluation of the expression.

```
datatype var_name <- initial_expression update: value_expression
[optional_attributes:...];
```

All the attributes of all the agents are updated at the same time, before they are given a chance to execute behaviors. Some examples of use for value:

- Automatically evolving attributes:

```
int my_int <- 0 update: my_int + 1: // -> my_int is incremented by 1 every
time step.
float my_float <- 100 update: my_float - (my_float / 100); // -> my_float
is decremented by 1% every time step.
```

- Sticky attributes:

```
int sticky_int update: 100; // -> whatever the changes made in the model to  
sticky_int, its value returns to 100 at the beginning of every step.
```

- Conditionally evolving attributes:

```
int cond_int update: (my_int < 100) ? 0 : my_int / 10; // -> the value of  
cond_int depends on that of my_int.  
float log_my_int update: ln(my_int); // -> the value of "cond_int" is  
always coupled to that of my_int.
```

Special facets

function

The update facet is computed only every step. But sometimes, we need more accurate updates (i.e. that the value of the attribute be evaluated each time we use it). The function facet has been introduced to this purpose and has the following syntax:

```
type1 var1 function: {an_expression} [optional_attributes:...];
```

Once a function is declared, whenever the attribute is used somewhere, the function is computed (so the value of the attribute always remains accurate). The declaration of function is **incompatible** with both init or update (an error will be raised). A shortcut has also been introduced:

```
type1 var1 -> {an_expression} [optional_attributes:...];
```

max, min

These two facets are only available in the context of int or float attributes. They allow the modeler to control the values of the attribute, by specifying a maximum and minimum value. The attribute will never be allowed to be lower than the minimum or greater than the maximum.

```
int max_energy <- 300 min: 100 max: 3000;
```

min and max combine gracefully with the parameter facet and allow to control what the user can enter, or the limits between which exploring the values of variables.

<_type_>

Only defined in the context of containers: matrix, list, map and graph attributes. Allows to define the type/species of values contained in the container. For instance, it can be handy, sometimes, to fix the species of the agents in a list at once rather than having to use the of_species operator every time. An example of that with the re-declaration of the built-in neighbours variable in a model with only one species of agents:

```
list<bug> neighbours;
```

Doing so enables the use of neighbours, in the following expressions, without having to specify which kind of agents are manipulated in it.

Temporary variable

A temporary variable (or local variable) is a variable that has an existence only in a statement block: as soon as the end of the block, the variable is deleted from the computer memory. It is local to the scope in which it is defined. The naming rules follow those of the variable declarations. In addition, a temporary variable cannot be declared twice in the same scope. The generic syntax is:

```
datatype temp_var1 <- an_expression;
```

After it has been declared this way, a temporary variable can be used like regular variables (for instance, the `<-` statement should be used to assign it a new value within the same scope).

—

Naming variables

Reserved Keywords

In GAML, some keywords are already reserved with a specific meaning and cannot be used for naming variables (and also species, actions, etc.). They are :

- The names of the global built-in variables
- The names of the primitive data types and new species defined in the model.
- The special keywords used by the language.
- The names of the variables found in every species.
- The names of the variables defined in skills when a species declares their use.
- The names of the units that can attached to numeric values.

Naming conventions

A variable name can be sequence of alphanumeric characters (plus the underscore, " `_` "). It should follow certain rules:

- it should not begin by a digit;
- it should not contain space characters.

By convention, it is recommended that:

- variable name begins by a lower case letter.

3.9 Defining Actions

Defining Actions (Under Construction)

```
<font color="red"> todo :  
  
• give more examples of various action calls and definitions  
• make a complete section on primitives (and how they handle their arguments)  
  
</font>
```

—
An action is a capability available to the agents of a species (what they can do). It is a block of statements that can be used and reused whenever needed. An action can accept arguments. An action can return a result (statement **return**).

Declaring Actions

From a general point of view, an action is declared with:

```
return_type action_name (var_type arg_name, ...) {  
    [sequence_of_statements]  
    [return value;]  
}
```

If an action does not return a value, the keyword **action** is to be used instead of the **return_type** .

```
action dummy_void {  
    write "dummy_void";  
}  
action dummy_void_arg(int c) {  
    write "dummy_void: " + c;  
}  
//in this action, the argument *a* could be omitted when calling the action  
(and will have *100* for default value)  
int dummy_return (int a <- 100, int b) {  
    return a + b;  
}
```

Some actions, called primitives, are directly coded in Java : for instance, the write action defined for all the agents.

Calling Actions

There are two ways to call an action: using a statement or as part of an expression

- action that does not return a result:

```
do dummy_void;  
do dummy_void_arg(10);
```

- action that returns a result:

```
my_var <- dummy_return (10, 50);  
my_var <- dummy_return (a: 10, b: 50);  
my_var <- dummy_return (b: 50);
```

It is possible to precise the agent that is calling the action:

```
my_var <- self dummy_return (10, 50);
```

The **self** keyword denotes the agent that will execute the action (the action must be defined in its species).

3.10 Defining Behaviors

Defining Behaviors (Under Construction)

```
<font color="red"> todo :  
    • give some insight on the execution order of reflexes  
    • link to the built-in architectures  
</font>
```

—
All agents (including the world and grid cells) are provided with a simple behavioral structure, based on reflexes. Species can define any number of reflexes within their body. In addition, all agents are provided with an init block that is activated at the creation of the agent is created.

reflex

A reflex is a sequence of statements that can be executed, at each time step, by the agent. If no attribute when are defined, it will be executed every time step. If there is an facet **when**, it is executed only if the boolean expression evaluates to true. It is a convenient way to specify the behavior of the agents. Example:

```
reflex my_reflex { //Executed every time step  
    write "Executing the unconditional reflex";  
}
```

```
reflex my_reflex when: flip (0.5){ //Only executed when flip returns true  
    write "Executing the conditional reflex";  
}
```

Init

A special form of reflex that is evaluated only once when the agent is created, after the initialization of its variables, and before it executes any reflex. Only one instance of init is allowed in each species

(except in case of inheritance, see this section). Useful for creating all the agents of a model in the definition of the world, for instance. Example:

```
init {
    color <- rnd_color(255);
}
```

3.11 Defining Equations

Defining equations (Under Construction)

 todo :

- proofread the language (difficult to read sometime)
- give more examples and explain the meaning of the "built-in" equations
- link with the recipe on how to use equations

—
This extension of GAMA is some new statements that provides new way to write Math models (System of equations) and new action to solve them with provided method (RK4, Dormand,...). This extension is implemented in the ummisco.gaml.extensions.maths plug-in.

Equations

Define equation(s)

Example of declaration:

```
species Maths {
    float t;
    float S;
    float I;
    float R;
    equation SIR {
        diff(S,t) = (- beta * S * I / N);
        diff(I,t) = (beta * S * I / N) - (alpha * I);
        diff(R,t) = (alpha * I);
    }
    ...
}
```

diff keyword, for instant, is a place-holder for a syntax more Mathematically. Its first parameter is the variable, the second (variable t) will not change during solve process.

Optional mode

simultaneously : [list of agents]

- simultaneously mode which accept a list of agent contains equations. The Solve statement will compute simultaneously at same time all the equations

Example of declaration:

```
species S_Maths {
    float S;
    equation eq simultaneously: [ I , first ( R ) ]{
        diff(self.S,t) = (- beta * self.S * first(I).size / N);
    }
    ...
}
```

This will take all equations in list of agents I and the first agent R, to make a complete system of equations.

action

solve

solve all equations which matched the name, in list of agent with given method and step. there are 3 optional variable could be declared inside

- **method** : string, mandatory, method of integration : "rk4" (only one, for instant).
- **step** : float, mandatory, step of integration.
- **t0** : float, optional, time initial of integration, default value: cycle-cycle_length.
- **tf** : float, optional, time final, default value: cycle.
- **cycle_length** : float, optional, length of simulation's cycle .

```
solve SIR method: "rk4" step:0.001{
    float cycle_length<-1.0;
    float t0<-cycle-cycle_length;
    float tf<-cycle;
}
```

Classical Equations

A set of classical ODEs is also provided (e.g. SIR, SI...).

Model examples

Please find some toy models which are defined in the folder models of this plugin on SVN.

3.12 Defining Aspects

Defining Aspects (Under Construction)

 todo :

- Explain that aspect can contain any other statements (not just draw) and give an example of a complex aspect
- begin the examples progressively (i.e. with simple, evident aspects)
- link with °pixel somewhere (and explain its meaning in the context of an aspect)

The aspect statement allows to define how the agents will be displayed. It is possible to define different displays (i.e. different aspect sections) for a same species. In this context, the user will be able to change the display drawn during the simulation execution.

Draw Command

The command **draw** allows to draw a geometry (line, circle, square, agent geometry...), a icon or a text:

- **geometry**: any arbitrary geometry, that will only be affected by the color facet.
- **text**: string, the text to draw.
- **image**: file, optional, path of the icon to draw (JPEG, PNG, GIF).

Several facets can be added:

- **color** : rgb, optional, the color to use to display the text/icon/geometry.
- **size** : float, size of the text/icon (not use in the context of the drawing of a geometry).
- **at** : point, location where the shape/text/icon is drawn.
- **rotate** : int, orientation of the shape/text/icon.
- **depth** : float, optional (only works if the type of the display is opengl). Add a depth to the geometry previously defined (a point becomes a sphere, a line becomes a plan, a circle becomes a cylinder, a square becomes a cube,a polygon becomes a polyhedron with depth equal to the z value). Note: This only works if the agent geometry is not a point.
- **border** : rgb, optional, the color to use to display the envelope of the geometry.

For example, the following model defines three aspects for the agent: one named "info", another named "icon" and the last one named "default".

```

aspect info {
    draw square(2) at: location rotate: heading;
    draw square(2) rotated_by heading;
    draw line([location, destination + (destination - location)]) to:
color:rgb("white") ;
    draw circle(4) at: location empty:true color:rgb("white") ;
    draw heading color: rgb("white") size:1;
    draw state color: rgb("white") size:1 at:my location + {1,1};
}

aspect icon {
    draw file_icon at: location size: 2 rotate: heading;
}
aspect default {
    draw square(2) at: location empty: !hasFood color:rgb("yellow") rotate:
heading;
}

```

—

Shape Primitive

GAMA provides several shape primitives to display agents.

Point	Line	Circle	Square	Polygon	Bitmap	GIS	
2D	+	/					
	Sphere	Plan	Cylinder	Cube	Polyhedron	3D Object	3D GIS
3D							

Examples of GAMA Shape Primitives with on [Gama3D16 opengl display]:

```

draw circle(10) color: rgb('red') at: {0,0,0};
draw cylinder(10,5) color: rgb('red') at: {25,0,0};

```

```
draw sphere(10) color: rgb('red') at: {50,0,0};  
draw square(20) color: rgb('blue') at: {0,25,0};  
draw cube(20) color: rgb('blue') at: {25,25,0};  
draw rectangle(20, 15) color: rgb('green') at: {0,50,0};  
draw rectangle({20, 15}) color: rgb('green') at: {25,50,0};  
draw box({20, 15 , 10}) color: rgb('green') at: {50,50,0};  
draw polygon ([{0,0},{10,0},{10,10},{15,15},{10,15},{5,10},{0,0}]) color:  
rgb('yellow') at: {0,75,0};  
draw polyhedron ([{0,0},{10,0},{10,10},{15,15},{10,15},{5,10},{0,0}],10)  
color: rgb('yellow') at: {25,75,0};  
draw line ([{0,0},{10,10}]) at: {0,100,0};  
draw plan ([{0,0},{10,10}],10) at: {25,100,0};  
draw polyline ([{0,0},{5,5},{10,0}]) at: {0,125,0};  
draw polyplan ([{0,0},{5,5},{10,0}],10) at: {25,125,0};
```

3.13 Defining User Commands

Defining User Commands

— 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.

Syntax

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::val1,  
arg2::val2, ...];
```

or

```
user_command kill_myself {  
    do die;  
}
```

These commands (which belong to the "top-level" statements like actions, reflexes, etc.) are not executed when an agent runs. Instead, they are collected and used as follows:

- When defined in a GUI experiment, they appear as buttons above the parameters of the simulation;
- When defined in the global block or in any species,
 - when the agent is inspected, they appear as buttons above the agents' attributes
 - 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];
    }
}
```

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 operator

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, 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 like the following one to force the user to input new values instead of relying on the initial values of parameters :

```
global {
    init {
```

```
    map values <- user_input([ "Number" :: 100]);
    create my_species number : int(values at "Number");
}
}
```

4. Defining GUI Experiments

Defining Experiments (under construction)

```
<font color="red"> todo :  
    • link to the UI of experiments  
    • explain that behaviors, __step__, __init__ are possible to define (and give examples)  
</font>
```

—
A GUI experiment allows to display a graphical interface with input parameters and outputs (display, file, monitor...). A GUI experiment is defined by:

```
experiment exp_name type: gui {  
    [input]  
    [output]  
}
```

4.1 Defining Parameters

Defining Parameters (Under Construction)

```
<font color="red"> todo :  
  
• link with the UI (Parameters View)  
• improve this definition: add the different facets allowed (among, min, max, etc.)  
• be more precise about the vocabulary used (i.e. not "global variable" but "attribute of the simulation agent")  
• give different examples and constraints (not 2 parameters with the same name, not 2 with the same var).  
  
</font>
```

— Experiments can define input, i.e. parameters. Defining parameters allows to make the value of a global variable definable by the user through the user graphic interface. A parameter is defined as follow:

```
parameter title var: global_var category: cat;
```

With:

- title: string to display
- var: reference to a global variable (defined in the global section)
- category: string used to «store» the operators on the UI (optional)

Example:

```
parameter "Value of toto: " var: toto;
```

4.2 Defining Outputs

Defining Outputs (Under Construction)

 todo :

- explain that output can also contain file statements, monitors and inspectors (not only display)
- explain the difference between 'output' and 'permanent'
- explain that the 'context', in output, is that of the simulation (i.e. no need to call "simulation.var" each time)

—
Output blocks define how to visualize a simulation (with one or more display blocks that define separate windows)

```
experiment exp_name type: gui {  
    [input]  
    output {  
        [display statements]  
        [monitor statements]  
        [file statements]  
    }  
}
```

4.2.1 Defining Displays

Defining Displays

A display refers to a independent and mobile part of the interface that can display species, images, texts or charts. The general syntax is:

```
display my_display [additional options] { ... }
```

Additional options include:

- **background** (type = color): the color of the display background
- **type** (2 possible values: java2D or opengl): specify if the display will use the java 2D ou openGL libraries. Note that the openGI display does not admit charts. The default value is java2D.
- **refresh_every** (type = int): the display will be refreshed every nb steps of the simulation
- **autosave** (type = boolean or location): if the value is true or is a location, GAMA will take each a snapshot of the display every time the display is refreshed. The location will precise the dimension of the picture.

There exist several kinds of display:

- classical **displays** (without specific type) used to species, text, image, charts...

```
display my_display { ... }
```

- **opengl displays** (display with type: opengl) used to display species, text or image. It allows to display 3D models.

```
display my_display type: opengl { ... }
```

Each display can be refreshed independently by defining the facet **refresh_every: nb (int)** (the display will be refreshed every nb steps of the simulation) Each display can include different layers (like in a GIS). Although every combinaison of any number of following layers are allowed in GAML, it is recommended to distinguish displays with species, image and/or text and display with charts (and text).

4.2.1.1 Defining Chart Layers

Chart Layer (Under construction)

```
display chart_display [additional options] {
    chart "chart name" type: series {
        data data1 value: mydata1 color: rgb('blue') ;
        data data2 value: mydata2 color: rgb('blue') ;
    }
}
```

Chart type

chart allows modeler to display a chart: GAMA can display 3 main types of charts using the **type** facet:

- histograms (**histogram**)
- pie (**pie**)
- series/xy/scatter: both display series with lines or dots, with 3 subtypes :
 - series (**series**) : to display the evolution of one/several variable (vs time or not).
 - xy (**xy**) : to specify both x and y value. To allow stacked plots, only one y value for each x value.
 - scatter (**scatter**) : free x and y values for each serie.

chart options include:

- **axes** optional, expects a rgb - the axis color
- **background** optional, expects a rgb - the background color
- **position** optional, expects a 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 only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- **size** optional, expects a 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** optional, expects an identifier, takes values in {stack, 3d, bar, exploded}. - No documentation yet
- **timexseries** optional, expects a list - for series charts, change the default time serie (simulation cycle) for an other value.

- **transparency** optional, expects a float - the style of the chart
- **x_range** , optional, expects any type in [a float, a int, a point] - 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_tick_unit** , optional, expects a float - the tick unit for the y-axis (distance between horyzontal lines and values on the left of the axis).
- **y_range** , optional, expects any type in [a float, a int, a point] - 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_tick_unit** , optional, expects a float - the tick unit for the x-axis (distance between vertical lines and values bellow the axis).

Data definition

Data can be specified with:

- several **data** statements to specify each serie
- one **datalist** statement to give a list of series. It can be useful if the number of series is unknown, variable or too high.

data statement

One data statement is used for one serie.

```
chart "DataBar" type:histogram
{
    data "empty_ants" value:(list(ant) count (!each.hasFood)) color:°red;
    data "carry_food_ants" value:(list(ant) count (each.hasFood)) color:
°green;
}
```

Only the value facet is required to provide the data value. It can be a single value (for histograms/pies/temporal series), lists (for series), points (for xy/scatter) or list of points (for scatter). Other options include:

- **color** , expects a rgb
- **fill** , expects a bool, to specify is the marker is filled or not
- **line_visible** , expects a bool, to specify if the line is visible or not (set to false to get a classic scatter plot)
- **marker** , expects a bool, to specify if the marker is visible or not
- **marker_shape** , takes values in {marker_square, marker_left_triangle, marker_right_triangle, marker_diamond, marker_hor_rectangle, marker_hor_ellipse, marker_vert_rectangle, marker_empty, marker_up_triangle, marker_sqaure, marker_down_triangle}.
- **style** , takes values in {exploded, dot, bar, step, area_stack, whisker, ring, line, 3d, spline, area, stack}. Available styles depend on the chart style (see bellow for each chart style).

datalist statement

One datalist statement is used for all the series.

```
chart "DataListBar" type:histogram
{
    datalist ["empty", "carry"] value:[(list(ant) count (!each.hasFood)),
(list(ant) count (each.hasFood))] color:[^red, ^green];
}
```

Only the value facet is required to provide the data values. It can be:

- a list of numbers (for histograms/pies/temporal series)
- a list of list of numbers (for histograms/series)
- a list of points/list with two numbers (for temporal xy/scatter)
- a list of list of points/lists with two numbers (for xy/scatter)

The number of series and values per serie (categories) doesn't have to be fixed, and series/categories names can change at each step. Series and categories names are set by :

- **categoriesnames** facet, with a list of string
- **legend** facet, with a list of string

Since you often have list of lists, it is possible to reverse categories and series using the **inverse_series_categories** keyword. For example `inverse_series_categories=true` will transform a value of `[[1,2,3],[4,5,6]]` to `[[1,4],[2,5],[3,6]]`. It May be useful when it is easier to construct one list over the other. Other options include:

- **color** , expects a list of colors
- **fill** , optional, expects a bool - same as data statement
- **line_visible** , optional, expects a bool - same as data statement
- **marker** , optional, expects a bool - same as data statement
- **style** , optional, expects an identifier, takes values in {exploded, dot, bar, step, area_stack, whisker, ring, line, 3d, spline, area, stack}. - same as data statement

Series/xy/scatter charts details

For series/xy/scatter charts

- With data statement, if each value is a single number, the value will be added to the chart at each chart step. If the value is a list, the value will be replaced by the list at each step.
- With datalist statement, if each value is a list of numbers, the value will be added to the chart at each chart step. If the value is a list of lists, the value will be replaced by the list at each step.

For example, this chart displays the positions of all the ants at each step (with two series):

```
display ChartScatterList {
    chart "DataListScatter" type:scatter
```

```
{  
    datalist ["empty", "carry"] value:[((list(ant) where (!  
each.hasFood)) collect each.location),((list(ant) where (each.hasFood))  
collect each.location)] color:[°red, °green] line_visible:false;  
  
}  
}
```

This one will display the history of the "average" ant with and without food:

```
display ChartScatterHistory {  
    chart "DataListScatterHistory" type:scatter  
    {  
        datalist ["empty", "carry"] value:[mean((list(ant) where (!  
each.hasFood)) collect each.location),mean((list(ant) where (each.hasFood))  
collect each.location)]  
            color:[°red, °green] line_visible:true;  
    }  
}
```

4.2.1.2 Defining Other Layers

Defining Other Layers

—
—

agents layer

agents allows the modeler to display only the agents that fulfill a given condition.

```
display my_display {
    agents layer_name value: expression [additional options];
}
```

Additional options include:

- **value** (type = container) the set of agents to display
- **aspect** : the name of the aspect that should be used to display the species.
- **transparency** (type = float, from 0 to 1): the transparency rate of the agents (1 means no transparency)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the species is refreshed. (usefull in case of agents that do not move)
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying agents
- **focus** (type = agent)

For instance, in a segregation model, agents will only display happy agents:

```
display Segregation {
    agents agentDisappear value : people as list where (each.is_happy =
false) aspect: with_group_color;
}
```

—

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. The general syntax is:

```
display display_name {  
    species species_name [additional options];  
}
```

Additional options include:

- **aspect** : the name of the aspect that should be used to display the species.
- **transparency** (type = float, from 0 to 1): the transparency rate of the agents (1 means no transparency)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the species is refreshed. (usefull in case of agents that do not move)
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying agents

For instance it could be:

```
display my_display{  
    species agent1 aspect: base ;  
}
```

Species can be superposed on the same plan:

```
display my_display{  
    species agent1 aspect: base;  
    species agent2 aspect: base;  
    species agent3 aspect: base;  
}
```

Species can be placed on different z values for each layer using the opengl display. z:0 means the layer will be placed on the ground and z=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 z:0;  
    species agent2 aspect: base z:0.5;  
    species agent3 aspect: base z:1;  
}
```

image layer

image allows modeler to display an image (e.g. as background of a simulation). The general syntax is:

```
display display_name {
    image layer_name file: image_file [additional options];
}
```

Additional options include:

- **file** (type = string): the name/path of the image (in the case of a raster image)
- **gis** (type = string): the name/path of the shape file (to display a shapefile as background, without creating agents from it)
- **color** (type = color): in the case of a shapefile, this the color used to fill in geometries of the shapefile
- **transparency** (type = float, from 0 to 1): the transparency rate (1 means no transparency)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **refresh** (type = boolean, **opengl only**): specify whether the display of the image is refreshed.
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying the image

For instance:

```
display my_display{
    image background file:"../images/my_backgound.jpg"
}
```

Or

```
display city_display refresh_every: 1 {
    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 type:opengl{
    image image1 file:"../images/image1.jpg";
    image image2 file:"../images/image2.jpg" z:0.5;
}
```

—

text layer

text allows the modeler to display a string (that can change at each step) in a given position of the display. The general syntax is:

```
display my_display {  
    text my_text value: expression [additional options];
```

Additional options include:

- **value** (type = string) the string to display
- **transparency** (type = float, from 0 to 1): the transparency rate of the layer (1 means no transparency)
- **position** (type = point, from {1,1} to {0,0}): position of the upper-left corner of the layer (note that {0.5,0.5} refers to the middle of the display)
- **size** (type = point, from {1,1} to {0,0}): size of the layer ({1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer)
- **font** : the font used for the text
- **color** (type = color): the color used to display the text
- **refresh** (type = boolean, **opengl only**): specify whether the layer is refreshed.
- **z** (type = float, from 0 to 1, **opengl only**): altitude of the layer displaying text

For instance:

```
display my_display {  
    text agents value : 'Carrying ants : ' + string ( int ( ant as list  
count (each . has_food ) ) + int ( ant as list count ( each . state =  
    'followingRoad' ) ) ) position : { 0.5 , 0.03 } color : rgb ( 'black' )  
size: { 1 , 0.02 };  
}
```

graphics layer

graphics allows the modeler to freely draw shapes/geometries/texts without having to define a species. The general syntax is:

```
display my_display {  
    graphics "my new layer" {  
        draw circle(5) at: {10,10} color: rgb("red");  
        draw "test" at: {10,10} size: 20 color: rgb("black");  
    }
```

4.2.1.3 3D Specific Instructions

3D Specific Instructions (Under construction)

—
—

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.

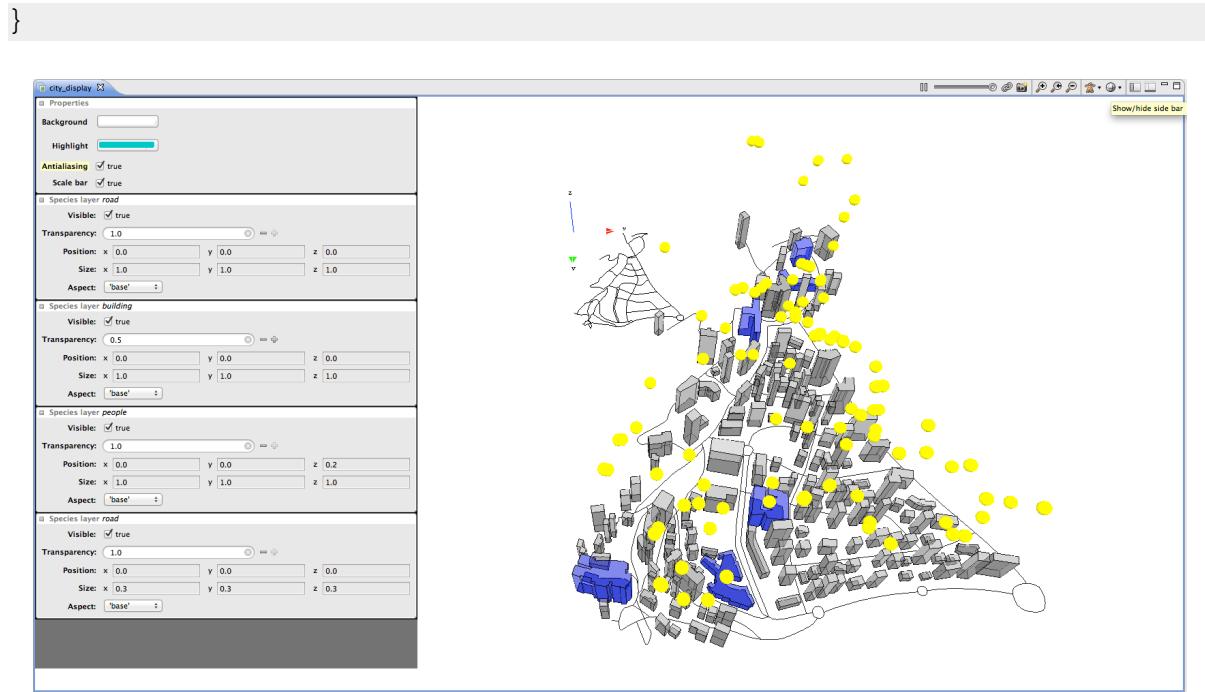
Position

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. You can also dynamically change those value by showing the side bar in the display.

```
display city_display type: opengl{
    species road aspect: base refresh:false;
    species building aspect: base transparency:0.5 ;
    species people aspect: base position:{0,0,0.2};
    species road aspect: base size:{0.3,0.3,0.3};
```



Camera

[G__ArcBallCamera Arcball Camera] [G__FreeFlyCamera 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

`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)

`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)

`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: <wiki:video url="<http://www.youtube.com/watch?v=lQVGD8aDKZY&feature=youtu.be>"> Boids 3D Visualization

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.

<wiki:video url="<http://www.youtube.com/watch?v=op56elmEEYs&feature=youtu.be>">

```
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)}{
...
}
```

—

4.2.2 Defining Inspectors and Monitors

Defining Inspectors and Monitors (Under Construction)

—

—

Monitors

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_every' facet). Definition of a monitor:

```
monitor monitor_name value: an_expression refresh_every: nb_steps;
```

with:

- value: mandatory, the expression whose value will be displayed by the monitor.
- refresh_every: int, optional : the number of simulation steps between two evaluations of the expression (default is 1).

Example:

```
monitor "nb preys" value: length(prey as list);
```

—

Inspectors

4.2.3 Defining Files

Defining Files

Saving data to a file during an experiment can 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 a `file` (or `*output_file*` statement in the output section.

```
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_every' 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).

5. Defining Batch Experiments

Defining Batch Experiments

Batch experiments allow 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]
}
```

The batch experiment facets

Batch experiments 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;  
        }  
    }  
}
```

```
    }  
}
```

5.1 Parameter Space

Parameter Space

Parameter definition

The **parameter** elements specifies which model parameters will change through the successive simulations. A parameter is defined as follows:

```
parameter title var: global_variable + possible_values
```

There are 2 ways to describe the range in which the value of the parameter will be explored :

- Explicit list: **among** : values_list

```
parameter "Value of toto:" var: toto among: [1, 3, 7, 15, 100];
```

- Range : **min** : min_value **max** : max_value **step** : increment_step

```
parameter "Value of toto:" var: toto min: 1 max: 100 step: 2;
```

For Strings and Booleans, you can only use the Explicit List. Each Batch methods may accept only some kind of definitions and parameter types. See the description of each of them for details.

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";
```

5.2 Exploration Methods

Exploration Methods

Several batch methods are currently available. Each is described below.

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;  
}
```

6. GAML Reference

Gaml Reference

The following pages introduce the various constructs that are made available to the modelers in GAML. All these constructs are **built-in**, i.e. defined in Java, either in the core of GAMA or in its extensions. They include [species](#), [skills](#), [control architectures](#), [statements](#), [data types](#), and various types of [expressions](#), including a long and quite exhaustive list of [operators](#). This set of pages is to be used as a *reference*, which means its role is not to explain *how* to properly use these constructs but to detail their structures and components. It is, in that respect, to be used as a support for following the explanations provided in the sections dedicated to the [organization of a model](#), the [definition of species](#), the definition of [GUI](#) or [batch](#) experiments, and the various recipes or how-to's provided [here](#). It also serves as a precious guide for the various constructs introduced in the [G__Tutorials tutorials].

6.1 Built-in Species

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;
```

The list of available built-in agents in GAMA is:

- cluster_builder: allows to use clustering techniques on a set of agents.
- multicriteria_analyzer: allows to use multi-criteria analysis methods.

[Top of the page](#)

Table of Contents

agent

Actions

init

- returns: unknown

step

- returns: unknown

[Top of the page](#)

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

helloWorld

- 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: list
- → **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

[Top of the page](#)

base_edge

Actions

[Top of the page](#)

experiment

Actions

[Top of the page](#)

graph_edge

Actions

[Top of the page](#)

—

graph_node

Actions

related_to

- returns: bool
- → **other** (agent):

[Top of the page](#)

—

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

[Top of the page](#)

—

multicriteria_analyzer

Actions

electre_DM

- returns: int

evidence_theory_DM

- returns: int

promethee_DM

- returns: int

weighted_means_DM

- returns: int

[Top of the page](#)

—

Physical3DWorld

Actions

computeForces

- returns: unknown

[Top of the page](#)

— 6.1.1 'agent'

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

6.1.2 'model'

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

— 6.1.3 'experiment'

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 [G__ExperimentSpecies `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

6.2 Built-in Skills

Built-in Skills

— This file is automatically generated from java files. Do Not Edit It.

— 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. The list of available skills in GAMA is:

- moving: for agents that need to move.

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 (r/o)" 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;
}
```

[Top of the page](#)

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 the agent follows
- **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 an 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 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 which 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)

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 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

speed_choice

action to choose a speed

- returns: float
- **new_road** (agent): the road on which to choose the speed

[Top of the page](#)

busTransportation

Variables

- **filePath** (string):
- **individualGraph** (graph):
- **isTemporalGraph** (boolean):
- **stationID** (string): DB identifier of the station

Actions

loadFile

moves the agent towards the target passed in the arguments.

- returns: void
- **source** (string): Path of the source file
- **datatype** (string): determine file datatype: OD - > it is an Origin Destination Matrix; busline - > official timetable of the transportation service

loadVehicleGraph

- returns: unknown
- **source** (graph):

travel_arrival

moves the agent towards the target passed in the arguments.

- returns: map
- **from** (string): departure station ID

- **to** (string): arrival Station ID
- **on** (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)
- **departureDate** (int): date of the departure

[Top of the page](#)

communicating

The communicating 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.
- **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 'refuse' 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

Reply a message with an 'end_conversation' pepformative 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

[Top of the page](#)

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): Weights 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): Weights 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)

[Top of the page](#)

—

driving2d

Variables

- **background_species** (list):
- **considering_range** (int):
- **currentDistance** (float):
- **currentPerimeter** (float):
- **isCalculatedPerimeter** (boolean):
- **isPassedFalseTarget** (boolean):
- **obstacle_species** (list):
- **speed** (float):

Actions

pedestrian_goto

moves the agent towards the target passed in the arguments.

- returns: int
- **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)
- **background** (list,agent,graph,geometry): list, agent, graph, geometry on which the agent moves (the agent moves inside this geometry)
- **on** (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)

read_replay

- returns: list
- **file_name** (string): File name.

read_replay_file

- returns: list
- **file_name** (string): File name.

save_replay

- returns: unknown
- **file_name** (string): File name.
- **agent_information** (list): list of position

vehicle_goto

moves the agent towards the target passed in the arguments.

- returns: int
- **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)
- **background** (list,agent,graph,geometry): list, agent, graph, geometry on which the agent moves (the agent moves inside this geometry)
- **on** (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)

[Top of the page](#)

graphic

The graphic skill is intended to define the minimal set of behaviours required from a graphical agent

Variables

- **transparency** (float): the transparency of the agent (between 0.0 and 1.0)

Actions

brewer_color

- returns: rgb
- **type** (string): Palette Type (Sequential, Diverging, Qualitative)
- **class** (int): Number of class
- **index** (int): index

brewer_palette

- returns: msj.gama.util. [GamaList]< java.awt.Color >
- **type** (string): Palette Type (Sequential, Diverging, Qualitative)

twinkle

- returns: void
- **period** (int): make the agent twinkle with a given period

[Top of the page](#)

—

grid

Variables

- **color** (rgb):
- **grid_value** (float):
- **grid_x** (int):
- **grid_y** (int):

Actions

[Top of the page](#)

—

humanmoving

Variables

Actions

approach

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **target** (agent):

approachAvoidPassedPosition

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **target** (agent):
- **passedList** (list):

approachAvoidPassedPosition2

- returns: point

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **target** (agent):
- **passedList** (list):

blindStraightWander

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **direction** (int):
- **target** (agent):

blindStraightWander2

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **direction** (int):
- **target** (agent):

blindWallTracking

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **passedList** (list):
- **target** (agent):

blindWallTracking2

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **passedList** (list):
- **target** (agent):

blindWander

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)

- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **target** (agent):

blindWander2

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** (agent):
- **target** (agent):

wanderAbove

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent

wanderAndAvoid

- returns: point
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **agent_size** (int): specification of size of the agent
- **background** ():
- **ignore_type** ():

[Top of the page](#)

MAELIA

Variables

Actions

maeliaTimeStamp

- returns: float

[Top of the page](#)

MDXSKILL

Variables

Actions

helloWorld

- returns: unknown

select

- returns: msi.gama.util. [GamaList]< java.lang.Object >
- **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

[Top of the page](#)

—

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): 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)

- **location** (point):
- **speed** (float): 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): 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)

[Top of the page](#)

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)

[Top of the page](#)

network

Variables

- **netAgtName** (string): Net ID of the agent

Actions

connectMessenger

moves the agent towards the target passed in the arguments.

- returns: void
- **to** (string): server URL
- **at** (string): server URL
- **withName** (string): agent Name

emptyMessageBox

moves the agent towards the target passed in the arguments.

- returns: bool

fetchMessage

moves the agent towards the target passed in the arguments.

- returns: map < string,unknown >

sendMessage

Send a message to a destination.

- returns: void
- **dest** (string): The network ID of the agent who receive the message
- **content** (any type): The content of the message

[Top of the page](#)

—

optimizing

Variables

Actions

computeAverageEvacuationTime

- returns: unknown

doTest

- returns: unknown

optimizeDirection

- returns: unknown

optimizeSigns

- returns: unknown

[Top of the page](#)

—

physical3D

Variables

- **collisionBound** (map):

- **density** (float):
- **mass** (float):
- **motor** (point):
- **physical_3D_world** (agent):
- **velocity** (list):

Actions

[Top of the page](#)

—

roadTrafficManagement

Variables

- **nbVehicle** (int):
- **speed** (float):

Actions

[Top of the page](#)

—

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** (agent): 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 \geq 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.

[Top of the page](#)

—

skill_road_node

Variables

- **block** (map): define the list of agents blocking the node, and for each agent, the list of concerned 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

[Top of the page](#)

—

socket

k

Variables

- **ip** (string): cs
- **msg** (string):
- **port** (int): t

Actions

listen_client

- returns: void

listen_server

- returns: void

open_socket

- returns: void

send_to_client

M.

- returns: void
- **msg** (string): td

send_to_server

M.

- returns: void
- **msg** (string): td

[Top of the page](#)

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

helloWorld

- returns: unknown

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: list
- **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

[Top of the page](#)

trafficMoving

Variables

Actions

goto_traffic

moves the agent towards the target passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: void
- **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)
- **duration** (float): duration of the moving
- **max_speed** (float): speedMoving
- **return_path** (boolean): if true, return the path followed (by default: false)

theoretical_duration

moves the agent towards the target passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: java.lang.Float
- **from** (point,geometry,agent): the location or entity towards which to move.
- **to** (point,geometry,agent): the location or entity towards which to move.
- **on** (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)
- **max_speed** (float): speedMoving

[Top of the page](#)

trafficMoving

Variables

Actions

goto_traffic

moves the agent towards the target passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: void
- **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)
- **duration** (float): duration of the moving
- **max_speed** (float): speedMoving
- **return_path** (boolean): if true, return the path followed (by default: false)

theoretical_duration

moves the agent towards the target passed in the arguments while considering the other agents in the network (only for graph topology)

- returns: java.lang.Float
- **from** (point,geometry,agent): the location or entity towards which to move.
- **to** (point,geometry,agent): the location or entity towards which to move.

- **on** (list,agent,graph,geometry): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)
- **max_speed** (float): speedMoving

[Top of the page](#)

6.3 Built-in Control Architectures

Built-in 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.

Attachment of Control Architecture

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 {  
}
```

List of Control Architectures

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 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.

6.3.1 Finite State Machine

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

```

6.3.2 Task Based

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
```

6.3.3 User Controlled

User Controlled

User controlled 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` .

Declaration

Using the Task architectures for a species require to use the **control** facet:

```
species user control: user_only {  
    ...  
}
```

```
species user control: user_first {  
    ...  
}
```

```
species user control: user_last {  
    ...  
}
```

If the control chosen is `user_first` , it means that the user controled 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 list(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 (ie. 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.

6.4 Statements

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 , chart , display_grid , display_population , event , graphics , image , overlay , quadtree , text ,
- **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 , pause_sound , release , resume_sound , run , setup , start_sound ,
stop_sound , switch , trace , transition , user_command , using ,
- **Single statement**
 - = , add , assert , break , data , datalist , diffusion , do , draw , error , export , let , put ,
remove , return , save , set , simulate , solve , user_input , warn , write ,
- **Species**
 - species ,
- **Variable (container)**
 - [#],
- **Variable (number)**
 - [#],
- **Variable (regular)**
 -

- [#],
 - **Variable (signal)**
 - signal ,
-

Statements by embedment

- **Behavior**
 - add , ask , capture , create , diffusion , do , error , if , let , loop , migrate , pause_sound , put , release , remove , resume_sound , return , run , save , set , simulate , solve , start_sound , stop_sound , switch , trace , transition , 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 ,
- **Layer**
 - draw , error , if , let , loop , switch , trace , warn , write ,
- **Model**
 - [#], [#], [#], action , aspect , equation , experiment , output , reflex , run , setup , species , state , task , test , user_command , user_init , user_panel ,
- **Sequence of statements or action**
 - add , ask , break , capture , create , data , datalist , diffusion , do , draw , error , if , let , loop , migrate , pause_sound , put , release , remove , resume_sound , return , save , set , simulate , solve , start_sound , stop_sound , switch , trace , transition , using , warn , write ,
- **Single statement**
 - run ,
- **Species**
 - [#], [#], [#], action , aspect , equation , plan , reflex , run , setup , signal , simulate , species , state , task , test , user_command , user_init , user_panel ,
- **action**
 - return ,
- **aspect**
 - draw ,
- **chart**
 - add , ask , data , datalist , do , put , remove , set , simulate , using ,
- **display**
 - agents , chart , display_grid , display_population , event , graphics , image , overlay , quadtree , text ,
- **display_population**
 - display_population ,
- **equation**
 - = ,

- **fsm**
 - `state` , `user_panel` ,
 - **if**
 - `else` ,
 - **output**
 - `display` , `inspect` , `monitor` , `output_file` ,
 - **permanent**
 - `display` , `inspect` , `monitor` , `output_file` ,
 - **state**
 - `enter` , `exit` ,
 - **switch**
 - `default` , `match` ,
 - **test**
 - `assert` ,
 - **user_command**
 - `user_input` ,
 - **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 *ommissible*). 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...;
}
```

[Top of the page](#)

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]):

Embedments

- The statement is of type: **Variable (number)**
- The statement can be embedded into: Species, Experiment, Model,
- The statement embeds statements:

[Top of the page](#)

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):
- 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
- 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):

Embedments

- The statement is of type: **Variable (regular)**
- The statement can be embedded into: Species, Experiment, Model,
- The statement embeds statements:

[Top of the page](#)

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):

Embedments

- The statement is of type: **Variable (container)**
- The statement can be embedded into: Species, Experiment, Model,
- The statement embeds statements:

[Top of the page](#)

Facets

- **right** (float), (omissible) : the right part of the equation (it is mandatory that it can be evaluate to 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)

Embedments

- The = statement is of type: **Single statement**
- The = statement can be embedded into: equation,
- The = statement embeds statements:

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](#) ,

[Top of the page](#)

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)

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](#) ,

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 between brackets 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](#) ,

[Top of the page](#)

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). Soon to be deprecated, please use 'add edge(..)' instead
- node (any type): an expression that will be added to a graph as a node. Soon to be deprecated, please use 'add node(..)' instead
- vertex (any type):
- weight (float):

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:

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](#) ,

[Top of the page](#)

—

agents

Facets

- **name** (a label), (omissible) : identifier of the layer
- **value** (container): the set of agents to display
- 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 with will be focus the camera (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 greater 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): 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)
- 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)

Embedments

- The agents statement is of type: **Layer**
- The agents statement can be embedded into: [display](#),
- The agents statement embeds statements:

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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

—

annealing

Facets

- name** (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the aggregation 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

Embedments

- The annealing statement is of type: **Batch method**
- The annealing statement can be embedded into: Experiment,
- The annealing statement embeds statements:

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;
```

[Top of the page](#)

—

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

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:

Definition

Allows an agent, the sender agent (that can be the [Sections161 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

- It obeys the following syntax, where the target facet denotes the receiver agent(s):

```
ask receiver_agent(s) {
    [statements]
}
```

- The species of the receiver agents must be known in advance for this statement to compile. If not, it is possible to cast them using the as facet. If the receiver_agent(s) is not instance(s) of the species a_species_expression, the execution will return a class cast exception in the set of statement:

```
ask receiver_agent(s) as: a_species_expression {
    [statement_set]
}
```

- As alternative form for the castin, if there is only a single receiver agent:

```
ask species_name (receiver_agent) {
    [statement_set]
}
```

- As alternative form for the castin, if receiver_agent(s) is a list of agents:

```
ask receiver_agents of_species species_name {
    [statement_set]
}
```

- 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
            neighbouring 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  
    }  
}  
}
```

[Top of the page](#)

—

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)

Embedments

- The aspect statement is of type: **Behavior**
- The aspect statement can be embedded into: Species, Model,
- The aspect statement embeds statements: **draw** ,

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;  
    }  
}
```

[Top of the page](#)

—

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: expresion

Embedments

- The assert statement is of type: **Single statement**
- The assert statement can be embedded into: test,
- The assert statement embeds statements:

Definition

Allows to check whether the evaluation of a given expression fulfils 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](#) ,

[Top of the page](#)

—

break

Facets

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:

Definition

break allows to interrupt the current sequence of statements.

Usages

[Top of the page](#)

—

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)

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:

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]).

```
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](#) ,

[Top of the page](#)

chart

Facets

- name** (a label), (omissible) : the identifier of the chart layer
- axes (rgb): the axis color
- background (rgb): the background color
- color (rgb):
- gap (float):
- 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 only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.

- 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: {exploded, 3d, stack, bar}:
- timexseries (list): for series charts, change the default time serie (simulation cycle) for an other value.
- transparency (float): the style of the chart
- type (an identifier), takes values in: {xy, scatter, histogram, series, pie, box_whisker}: the type of chart. It could be histogram, series, xy or pie. 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_range (any type in [float, int, point]): 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_tick_unit (float): the tick unit for the y-axis (distance between horyzontal lines and values on the left of the axis).
- y_range (any type in [float, int, point]): 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_tick_unit (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).

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** ,

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** ,

[Top of the page](#)

create

Facets

- species (species), (omissible) : an expression that evaluates to a species, the species of created agents
- as (species):
- from (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 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 lsit 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

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:

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)  
}
```

- Desprecated uses:

```
// Simple syntax
create species: a_species number: an_int;
```

- If number equals 0 or species is not a species, the statement is ignored.

[Top of the page](#)

data

Facets

- **value** (any type in [float, point, list]):
- legend (string), (omissible) :
- color (rgb):
- fill (boolean):
- line_visible (boolean):
- marker (boolean):
- marker_shape (an identifier), takes values in: {marker_empty, marker_sqaure, marker_square, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}:
- name (an identifier):
- style (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}:

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:

[Top of the page](#)

datalist

Facets

- **value** (list): the values to display. Has to be 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)
- categoriesnames (list): the name of categories (can be a variable with dynamic names)
- color (list): list of colors
- fill (boolean):
- inverse_series_categories (boolean): reverse the order of series/categories ([[1,2] , [3,4] , [5,6]] -- > [[1,3,5] , [2,4,6]]. May be useful when it is easier to construct one list over the other.
- line_visible (boolean):
- marker (boolean):
- style (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: series style

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:

[Top of the page](#)

—

default

Facets

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:

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](#) ,

[Top of the page](#)

—

diffusion

Facets

- **var** (an identifier), (omissible) : the variable to be diffused
- **on** (an identifier): the species (in general a grid), on which the diffusion will occur
- **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)
- **mat_diffu** (matrix): the diffusion matrix (can have any size)
- **method** (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- **proportion** (float): a diffusion rate
- **radius** (int): a diffusion radius

Embedments

- The diffusion statement is of type: **Single statement**
- The diffusion statement can be embedded into: Behavior, Sequence of statements or action,
- The diffusion statement embeds statements:

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]]);
diffusion var: phero on: cells mat_diffu: math_diff;
```

- The diffusion can be masked by obstacles, created from a bitmap image:

```
diffusion 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):

```
diffusion var: phero on: cells proportion: 1/9 radius: 1;
```

[Top of the page](#)

—

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(125,125,125)
- **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_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 rgb(125,125,125)
- **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 enable/disable the drawing of the diffuse light. 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 displau will (dynamically) focus
- **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
- **z_fighting** (boolean): Allows to alleviate a problem where agents at the same z would overlap each other in random ways

Embedments

- The display statement is of type: **Output**
- The display statement can be embedded into: output, permanent,
- The display statement embeds statements: `agents` , `chart` , `display_grid` , `display_population` , `event` , `graphics` , `image` , `overlay` , `quadtree` , `text` ,

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;
}
```

[Top of the page](#)

—

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, give 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 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)
- 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)
- 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]): the file object containing the texture image to be applied on the grid
- transparency (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)
- triangulation (boolean): specifies whether 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)

Embedments

- The display_grid statement is of type: **Layer**
- The display_grid statement can be embedded into: display,
- The display_grid statement embeds statements:

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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

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 greater 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)
- 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)
- 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)

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](#) ,

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](#) , [quadtree](#) , [text](#) ,

[Top of the page](#)

—

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

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:

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;  
    ...  
}
```

[Top of the page](#)

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):
- border (rgb): the colors of the geometry border
- color (rgb): the color to use to display the text/icon/geometry
- depth (float): (only if the display type is opengl) Add a 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 a the agent has not a point geometry
- 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 (string): the font used to draw the text
- image (string): path of the icon to draw (JPEG, PNG, GIF)
- rotate (any type in [float, int]): orientation of the shape/text/icon
- rounded (boolean): specify whether the geometry have to be rounded (e.g. for squares)
- scale (float):
- shape (any type): the shape to display
- size (float): size of the text/icon (not use in the context of the drawing of a geometry)
- style (an identifier), takes values in: {plain, bold, italic}: the style used to display text
- text (string): the text to draw
- texture (any type in [string, list]): the texture that should be applied to the geometry
- to (point):

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:

Definition

draw is used in an aspect block to express 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
size: 0.8 ;
    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;
}
```

[Top of the page](#)



else

Facets

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:

Definition

This statement cannot be used alone

Usages

- See also: [if](#),

[Top of the page](#)

—

enter

Facets

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:

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](#) ,

[Top of the page](#)

—

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 agents 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

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: [=](#) ,

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] 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](#),

[Top of the page](#)

error

Facets

- **message** (string), (omissible) : the message to display in the error.

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:

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

- Other examples of use:

```
error 'This is an error raised by ' + self;
```

[Top of the page](#)

event

Facets

- **name** (an identifier), (omissible) , takes values in: {mouse_up, mouse_down, mouse_drag}: the type of event captured
- **action** (string): the identifier of the action to be executed. It has to be an action written in the global block. This action have to follow the following specification:
action myAction (point location, list selected_agents)
- **mouse_location** (string):
- **selected_agents** (string):

Embedments

- The event statement is of type: **Layer**
- The event statement can be embedded into: display,
- The event statement embeds statements:

Definition

event allows to interact with the simulation by capturing mouse event and doing an action. This action could apply a change on environment or on agents, according to the goal.

Usages

- The general syntax is:

```
event [event_type] action: myAction;
```

- For instance:

```
global {
    // ...
    action myAction (point location, list 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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

—

exhaustive

Facets

- **name** (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the aggregation method
- maximize (float): the value the algorithm tries to maximize
- minimize (float): the value the algorithm tries to minimize

Embedments

- The exhaustive statement is of type: **Batch method**
- The exhaustive statement can be embedded into: Experiment,
- The exhaustive statement embeds statements:

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;
```

[Top of the page](#)

—

exit

Facets

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:

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](#) ,

[Top of the page](#)

experiment

Facets

- **name** (a label), (omissible) : identifier of the experiment
- **title** (a label):
- **type** (a label), takes values in: {batch, gui}: the type of the experiment (either 'gui' or 'batch')
- control (an identifier):

- frequency (int): the execution frequency of the experiment (default value: 1). If frequency: 10, the experiment is executed only each 10 steps.
- keep_seed (boolean):
- multicore (boolean):
- parent (an identifier): the parent experiment (in case of inheritance between experiments)
- repeat (int): In case of a batch experiment, expresses how 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 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:

[Top of the page](#)

—

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:

[Top of the page](#)

—

genetic

Facets

- **name** (an identifier), (omissible) :

- aggregation (a label), takes values in: {min, max}: the aggregation 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)

Embedments

- The genetic statement is of type: **Batch method**
- The genetic statement can be embedded into: Experiment,
- The genetic statement embeds statements:

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;
```

[Top of the page](#)

graphics

Facets

- **name** (a label), (omissible) : the identifier 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 greater 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)
- 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)
- 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)

Embedments

- The graphics statement is of type: **Layer**
- The graphics statement can be embedded into: display,
- The graphics statement embeds statements:

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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

hill_climbing

Facets

- **name** (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the aggregation 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

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:

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;
```

[Top of the page](#)

—

if

Facets

- **condition** (boolean), (omissible) : A boolean expression: the condition that is evaluated.

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`,

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]
}
```

[Top of the page](#)

—

image

Facets

- name (string), (ommissible) : the identifier 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 greater 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): 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)
- transparency (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Embedments

- The image statement is of type: **Layer**
- The image statement can be embedded into: display,
- The image statement embeds statements:

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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

inspect

Facets

- **name** (any type), (ommissible) : 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, species, display_population, 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

Embedments

- The inspect statement is of type: **Output**
- The inspect statement can be embedded into: output, permanent,
- The inspect statement embeds statements:

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"];
```

[Top of the page](#)

—

let

Facets

- **name** (a new identifier), (omissible) :
- **value** (any type):
- **index** (a datatype identifier):
- **of** (a datatype identifier):
- **type** (a datatype identifier):

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:

[Top of the page](#)

—

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

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:

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 or 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.

```
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
```

[Top of the page](#)

—

match

Facets

- value (any type), (omissible) :

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:

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](#) ,

[Top of the page](#)

migrate

Facets

- **source** (an identifier), (omissible) : can be an agent, a list of agents, a agent's population to be migrated
- **target** (an identifier): target species/population that source agent(s) migrate to.
- returns (a new identifier): the list of returned agents in a new local variable

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:

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](#) ,

[Top of the page](#)

monitor

Facets

- **name** (a label), (omissible) : identifier of the monitor
- **value** (any type): expression that will be evaluated to be displayed in the monitor
- **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)

Embedments

- The monitor statement is of type: **Output**
- The monitor statement can be embedded into: output, permanent,
- The monitor statement embeds statements:

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;
```

[Top of the page](#)

—

output

Facets

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](#) ,

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 or monitor statements]  
    }  
}
```

- See also: [display](#) , [monitor](#) , [inspect](#) , [output_file](#) ,

[Top of the page](#)

output_file

Facets

- **name** (an identifier), (omissible) :
- **data** (string):
- **footer** (string):
- **header** (string):
- **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):
- **type** (an identifier), takes values in: {csv, text, xml}:

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:

[Top of the page](#)

overlay

Facets

- **left** (any type), (omissible) : an expression that will be evaluated and displayed in the left section of the overlay
- **center** (any type): an expression that will be evaluated and displayed in the center section of the overlay
- **color** (any type in [list, rgb]): the color(s) used to display the expressions given in other facets
- **right** (any type): an expression that will be evaluated and displayed in the right section of the overlay

Embedments

- The overlay statement is of type: **Layer**

- The overlay statement can be embedded into: display,
- The overlay statement embeds statements:

Definition

overlay allows the modeler to display a line to the already existing overlay, where the results of 'left', 'center' and 'right' facets, when they are defined, are displayed with the corresponding color if defined.

Usages

- The general 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](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

—

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

Embedments

- The parameter statement is of type: **Parameter**
- The parameter statement can be embedded into: Experiment,
- The parameter statement embeds statements:

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;
```

[Top of the page](#)

pause_sound

Facets

Embedments

- The pause_sound statement is of type: **Sequence of statements or action**
- The pause_sound statement can be embedded into: Behavior, Sequence of statements or action,
- The pause_sound statement embeds statements:

[Top of the page](#)

permanent

Facets

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` ,

Definition

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;
        }
    }
}
```

[Top of the page](#)

plan

Facets

- name (an identifier), (omissible) :
- executed_when (boolean):
- priority (float):
- when (boolean):

Embedments

- The plan statement is of type: **Behavior**
- The plan statement can be embedded into: Species,
- The plan statement embeds statements:

[Top of the page](#)

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

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:

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]
```

[Top of the page](#)

—

quadtree

Facets

- **name** (a label), (omissible) : identifier of the layer
- **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 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)
- **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)
- **transparency** (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Embedments

- The quadtree statement is of type: **Layer**
- The quadtree statement can be embedded into: display,
- The quadtree statement embeds statements:

Definition

quadtree allows the modeler to display the quadtree.

Usages

- The general syntax is:

```
display my_display {
    quadtree 'qt' position: { 0, 0.5 } size: quadrant_size;
}
```

- See also: [display](#) , [agents](#) , [chart](#) , [event](#) , [graphics](#) , [display_grid](#) , [image](#) , [overlay](#) , [quadtree](#) , [display_population](#) , [text](#) ,

[Top of the page](#)

reactive_tabu

Facets

- **name** (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the aggregation 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

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:

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;
```

[Top of the page](#)

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.

Embedments

- The reflex statement is of type: **Behavior**
- The reflex statement can be embedded into: Species, Experiment, Model,
- The reflex statement embeds statements:

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";
}
```

[Top of the page](#)

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)

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:

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]). 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. They won't consider "B" agent as their macro-agent (host) anymore. Their host (macro-agent) will 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](#),

[Top of the page](#)

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. Soon to be deprecated, use 'remove edge(item)...' instead
- 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. Soon to be deprecated, use 'remove node(item)...' instead
- vertex (any type):

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:

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 occurrence 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 can 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 removed 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](#) ,

[Top of the page](#)

resume_sound

Facets

Embedments

- The resume_sound statement is of type: **Sequence of statements or action**
- The resume_sound statement can be embedded into: Behavior, Sequence of statements or action,
- The resume_sound statement embeds statements:

[Top of the page](#)

return

Facets

- value (any type), (omissible) : an expression that is returned

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:

Definition

Allows to specify which value to return from the evaluation of the surrounding statement. Usually used within the declaration of an action. For more details about actions, see the following [Section161 section].

Usages

- Contrary to other languages, using return does not stop the evaluation of the surrounding statement (for instance, a loop). It simply indicates what value to return: if it is inside a loop, then, only the last evaluation of return will be returned. 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"  
}
```

[Top of the page](#)

run

Facets

- **experiment** (string), (omissible) :
- **of** (string):
- **core** (int):
- **end_cycle** (int):
- **out** (string):
- **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:

[Top of the page](#)

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 projectcion, 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):

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:

Definition

Allows to save data in a file. The type of file can be "shp", "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 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";
```

- The save statement can be used in an init block, a reflex, an action or in a user command. Do not use it in experiments.

[Top of the page](#)

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:

[Top of the page](#)

set

Facets

- **name** (any type), (omissible) : the name of an existin variable or attribute to be modified
- **value** (any type): the value to affect to the variable or attribute

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:

Definition

Allows to assign a value to the variable or attribute specified

Usages

- Other examples of use:

[Top of the page](#)

setup

Facets

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:

Definition

The setup statement is used to define the set of instructions that will be executed before every [test](#) .

Usages

- As every test should be independant 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](#) ,

[Top of the page](#)

signal

Facets

- **name** (a new identifier), (omissible) : The name of the variable that will be introduced to represent this signal on the specified grid
- **decay** (float): represents the amount to remove to the intensity of a signal, once dropped on a place, at each time step. It is a percentage between 0 and 1. If 'decay' is not defined, the signal will not be wiped from the places; otherwise, its intensity will be equal to (intensity * decay).
- among (list):
- environment (species): The name of the grid species on which this signal will be propagated
- on (any type in [species, container]): Either the name of the grid species on which this signal will be propagated (equivalent to 'environment:'), or an expression that returns a subset of the cells of this grid species
- propagation (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagations of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbours 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 neighbours is : $S' = (S / N / \text{proportion}) - \text{variation}$. The intensity of S is then diminished by $S * \text{proportion}$ 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 neighbours receives the intensity : $S / \text{proportion} - \text{variation}$. If multiple propagations 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 value between 0 and 1 that represents the percentage of the intensity which will be shared between the neighbours in the diffusion. For instance, for an intensity of 80, and a proportion of 0.5, in a 4-neighbours environment, each of the neighbouring places will receive an intensity of $(80 * 0.5) / 4 = 10$. If no 'proportion' is defined, it is assumed that the propagation corresponds to a diffusion where 100% of the intensity is equally divided between the neighbours. I.e., for an intensity of 100, and 4 neighbours per place, each of them receives a signal with an intensity of 25.
- range (float): Indicates the distance (in meter) at which the signal stops propagating
- type (a datatype identifier):
- update (any type): An expression that will be evaluated each cycle to update the value of the signal on each grid cell
- value (any type):
- variation (float): an absolute decrease of intensity that occurs between each place. It should be a positive number. However, negative numbers are allowed (be aware, in this case, that if no range is defined, the signal will certainly propagate in the whole environment). If no 'variation' is defined, it defaults to 1 in the case of a gradient type and 0 in the case of a diffusion.

Embedments

- The signal statement is of type: **Variable (signal)**
- The signal statement can be embedded into: Species,
- The signal statement embeds statements:

[Top of the page](#)

simulate

Facets

- **comodel** (file), (omissible) :
- repeat (int):
- reset (boolean):
- share (list):
- until (boolean):
- with_experiment (string):
- with_input (map):
- with_output (map):

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:

Definition

Allows an agent, the sender agent (that can be the [Sections161 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]
}
```

[Top of the page](#)

solve

Facets

- **equation** (string), (omissible) : the equation system identifier to be numerically solved
- **step** (float): integration step, use with most integrator methods (default value: 1)
- **cycle_length** (int): length of simulation cycle which will be synchronize with step of integrator (default value: 1)
- **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
- **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: {rk4, dp853}: integrate method (can be only "rk4" or "dp853") (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)
- **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

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:

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;
```

[Top of the page](#)

species

Facets

- **name** (an identifier), (omissible) : the identifier of the species
- **compile** (boolean):
- **control** (a label): defines the architecture of the species (e.g. fsm...)
- **edge_species** (an identifier): 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.
- **neighbours** (int): (grid only), the chosen neighbourhood (4, 6 or 8)
- **parent** (an identifier): 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 (default: 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_neighbours_cache** (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbours 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)

Embedments

- The species statement is of type: **Species**
- The species statement can be embedded into: Model, Environment, Species,
- The species statement embeds statements:

Definition

The species statement allows modelers to define new species in the model. global and grid are special 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  
neighbours: 8 use_regular_agents: false {
```

- Using a file to initialize the grid can replace width/height facets:

```
grid ant_grid file: grid_file neighbours: 8 use_regular_agents: false {
```

[Top of the page](#)

start_sound

Facets

- **source** (string): The path to music file. This path is relative to the path of the model.
- mode (an identifier), takes values in: {overwrite, ignore}: Mode of
- repeat (boolean):

Embedments

- The start_sound statement is of type: **Sequence of statements or action**
- The start_sound statement can be embedded into: Behavior, Sequence of statements or action,
- The start_sound statement embeds statements:

[Top of the page](#)

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)

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](#) ,

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](#) ,

[Top of the page](#)

stop_sound

Facets

Embedments

- The stop_sound statement is of type: **Sequence of statements or action**
- The stop_sound statement can be embedded into: Behavior, Sequence of statements or action,
- The stop_sound statement embeds statements:

[Top of the page](#)

switch

Facets

- **value** (any type), (omissible) : an expression

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](#) ,

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](#) ,

[Top of the page](#)

—

tabu

Facets

- name** (an identifier), (omissible) :
- aggregation (a label), takes values in: {min, max}: the aggregation 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

Embedments

- The tabu statement is of type: **Batch method**
- The tabu statement can be embedded into: Experiment,
- The tabu statement embeds statements:

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;
```

[Top of the page](#)

—

task

Facets

- **name** (an identifier), (omissible) : the identifier of the task
- **weight** (float): the priority level of the task

Embedments

- The task statement is of type: **Behavior**
- The task statement can be embedded into: weighted_tasks, Species, Experiment, Model,
- The task statement embeds statements:

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

[Top of the page](#)

—

test

Facets

- name (an identifier), (omissible) : identifier of the test

Embedments

- The test statement is of type: **Behavior**
- The test statement can be embedded into: Species, Experiment, Model,
- The test statement embeds statements: [assert](#),

Definition

The test statement allows modeler to define a set of assertions that will be tested. Before the execution of the embeded 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](#) ,

[Top of the page](#)

—

text

Facets

- **name** (string), (omissible) : the string to display
- color (rgb): the color used to display the text

- font (an identifier): the font used for the text
- 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 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 text is refreshed. (true by default, useful in case of text that is not been modified over simulation)
- size (any type in [int, float, point]): the layer resize factor: {1,1} refers to the original size whereas {0.5,0.5} divides it by 2
- style (an identifier), takes values in: {plain, bold, italic}: the style (bold, italic...) used to display the text
- transparency (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)
- value (string):

Embedments

- The text statement is of type: **Layer**
- The text statement can be embedded into: display,
- The text statement embeds statements:

Definition

text allows the modeler to display a string (that can change at each step) in a given position of the display.

Usages

- The general syntax is:

```
display my_display {
    text expression [additional options];
}
```

- For instance, in a segregation model, agents will only display unhappy agents:

```
display Segregation {
    text 'Carrying ants : ' + (int(ant as list count(each.has_food)) +
    int(ant as list count(each.state = 'followingRoad'))) position: {0.5,0.03}
    color: rgb('black') size: {1,0.02};
}
```

- See also: [display](#) , [agents](#) , [chart](#) , [event](#) , [graphics](#) , [display_grid](#) , [image](#) , [overlay](#) , [quadtree](#) , [display_population](#) ,

[Top of the page](#)

trace

Facets

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:

Definition

All the statements executed in the trace statement are displayed in the console.

Usages

[Top of the page](#)

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

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:

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](#) ,

[Top of the page](#)

—

user_command

Facets

- **name** (a label), (omissible) : the identifier of the user_command
- **action** (an identifier): the identifier of the action to be executed
- **when** (boolean): the condition that should be fulfilled in order that the action is executed
- **with** (map): the map of the parameters::values that requires the action

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](#) ,

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](#) ,

[Top of the page](#)

user_init

Facets

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:

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](#) ,

[Top of the page](#)

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

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:

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](#) ,

[Top of the page](#)

—

user_panel

Facets

- **name** (an identifier), (omissible) :
- **initial** (boolean):

Embedments

- The user_panel statement is of type: **Behavior**
- The user_panel statement can be embedded into: fsm, Species, Experiment, Model,
- The user_panel statement embeds statements: [user_command](#) ,

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 an 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](#) ,

[Top of the page](#)

using

Facets

- **topology** (topology), (omissible) : the topology

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:

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);
}

```

[Top of the page](#)

warn

Facets

- **message** (string), (omissible) : the message to display as a warning.

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:

Definition

The statement makes the agent output an arbitrary message in the error view as a warning.

Usages

- Other examples of use:

```
warn 'This is a warning from ' + self;
```

[Top of the page](#)

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.

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:

Definition

The statement makes the agent output an arbitrary message in the console.

Usages

- Other examples of use:

```
write 'This is a message from ' + self;
```

[Top of the page](#)

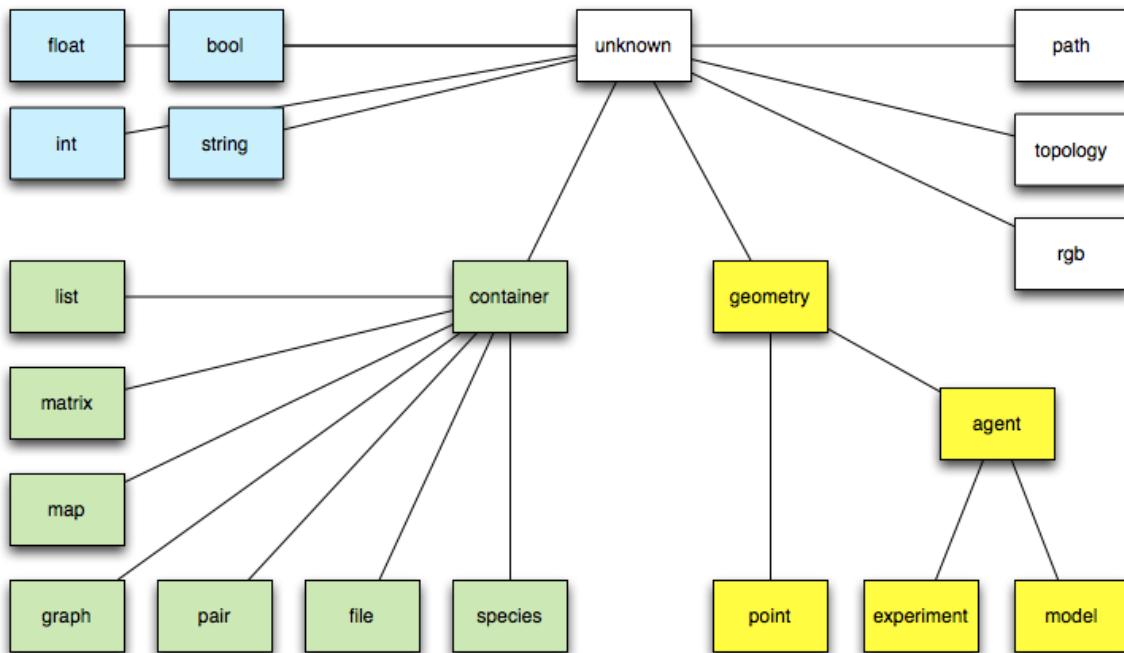
6.5 Data Types

Types (Under Construction)

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-dependant) is the following:



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 comprised between -(2⁻²⁵²) * 2¹⁰²³ and -(2⁻²⁵²) * 2¹⁰²³.
- **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 -2³¹ and 2³¹ - 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 [Operators_14 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:** [Operators_14 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:** [Operators_14 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:** [Operators_14 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:** [Operators_14 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[], []]
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 litteraly; 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 <- list [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.
- **Literal 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).
- **Literal 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 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 lines, 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 lines, 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 possibility
                                         // 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.
- **Literal 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.
- **Literal 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 components 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 neighbourhoods, 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 topology. 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...)
- **See also:** [Operators_14 Topology operators]
- **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 neighbourhood and distances
 - a geometry graph: returns a graph topology which computes specifically neighbourhood 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](#)

6.5.1 File Types

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.

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
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<geometry> osmfile <- file<geometry> (osm_file("../includes/rouen.gz",
filtering)) ;
```

- **osm_file(string path, map filter, 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, only the elements with the defined values are loaded from the file.
- **osm_file(string path, map filter, 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, only the elements with the defined values are loaded from the file.
- **is_osm(op)** : tests whether the operand is a osm file

Grid File

Esri ASCII Grid files are classic text raster GIS data. More details about Esri ASCII grid file can be found [here](#). Note that grid files can be used to initialize a grid species. The number of rows and columns will be read from the file. Similarly, the values of each cell contained in the grid file will be accessible through the **grid_value** attribute.

```
grid cell file: grid_file {  
}
```

Extensions

Here the list of possible extension for grid file:

- "asc"

Content

The content of a grid file is a list of geometries corresponding to the cells of the grid. For example:

```
global {
    init {
        file my_file <- grid_file("../includes/data.asc");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
...
...
```

Note that the values of each cell of the grid file is stored in their corresponding GAMA geometry (**grid_value** attribute). The operator "get" (or "read") allows to get the value of this attribute. For example:

```
file my_file <- grid_file("../includes/data.asc");
write "my_file: " + my_file.contents;
loop el over: my_file {
    write el get "grid_value";
}
```

Operators

List of operators related to shapefiles:

- **grid_file(string path)** : load a file (with an authorized extension) as a grid file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference).
- **grid_file(string path, string code)** : load a file (with an authorized extension) as a grid file with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: <http://spatialreference.org/ref/>)

- **grid_file(string path, int EPSG_ID)** : load a file (with an authorized extension) as a grid file 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 <- grid_file("../includes/data.shp", "EPSG:32601");
```

- **is_grid(op)** : tests whether the operand is a grid file.

—

Image File Extensions

Here the list of possible extensions for image file:

- "tif"
- "tiff"
- "jpg"
- "jpeg"
- "png"
- "gif"
- "pict"
- "bmp"

Content

The content of an image file is a matrix of int: each pixel is a value in the matrix. For example:

```
global {
    init {
        file my_file <- image_file("../includes/DEM.png");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
-9671572
-9671572
-9671572
-9671572
-9934744
-9934744
-9868951
```

```
-9868951
-10000537
-10000537
...
```

Operators

List of operators related to csv files:

- **image_file(string path)** : load a file (with an authorized extension) as an image file.
 - **image_file(string path, matrix content)** : load a file (with an authorized extension) as an image file and fill it with the given content.
 - **is_image(op)** : tests whether the operand is an image file
-

SVG File

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. Note that interactivity and animation features are not supported in GAMA. More details about SVG file can be found [here](#).

Extensions

Here the list of possible extension for SVG file:

- "svg"

Content

The content of a SVG file is a list of geometries. For example:

```
global {
    init {
        file my_file <- svg_file("../includes/data.svg");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
```

Operators

List of operators related to svg files:

- **shape_file(string path)** : load a file (with an authorized extension) as a SVG file.
- **shape_file(string path, point size)** : load a file (with an authorized extension) as a SVG file with the given size:

```
file my_file <- svg_file("../includes/data.svg", {5.0,5.0});
```

- **is_svg(op)** : tests whether the operand is a SVG file

Property File

Extensions

Here the list of possible extensions for property file:

- "properties"

Content

The content of a property file is a map of string corresponding to the content of the file. For example:

```
global {
    init {
        file my_file <- property_file("../includes/data.properties");
        loop el over: my_file {
            write el;
        }
    }
}
```

with the given property file:

```
sepallength = 5.0
sepalwidth = 3.0
petallength = 4.0
petalwidth = 2.5
type = Iris-setosa
```

will give:

```
3.0
4.0
```

```
5.0
Iris-setosa
2.5
```

Operators

List of operators related to text files:

- **property_file(string path)** : load a file (with an authorized extension) as a property file.
- **is_property(op)** : tests whether the operand is a property file

—

R File

R is a free software environment for statistical computing and graphics. GAMA allows to execute R script (if R is installed on the computer). More details about R can be found [here](#). Note that GAMA also integrates some operators to manage R scripts:

- [R_compute](#)
- [R_compute_param](#)

Extensions

Here the list of possible extensions for R file:

- "r"

Content

The content of a R file corresponds to the results of the application of the script contained in the file.
For example:

```
global {
    init {
        file my_file <- R_file("../includes/data.r");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
3.0
```

Operators

List of operators related to R files:

- **R_file(string path)** : load a file (with an authorized extension) as a R file.
- **is_R(op)** : tests whether the operand is a R file.

—

3DS File

3DS is one of the file formats used by the Autodesk 3ds Max 3D modeling, animation and rendering software. 3DS files can be used in GAMA to load 3D geometries. More details about 3DS file can be found [here](#).

Extensions

Here the list of possible extension for 3DS file:

- "3ds"
- "max"

Content

The content of a 3DS file is a list of geometries. For example:

```
global {
    init {
        file my_file <- threeds_file("../includes/data.3ds");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
```

Operators

List of operators related to 3ds files:

- **threeds_file(string path)** : load a file (with an authorized extension) as a 3ds file.
- **is_threeds(op)** : tests whether the operand is a 3DS file

OBJ File

OBJ file is a geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors. More details about Obj file can be found [here](#) .

Extensions

Here the list of possible extension for OBJ files:

- "obj"

Content

The content of a OBJ file is a list of geometries. For example:

```
global {
    init {
        file my_file <- obj_file("../includes/data.obj");
        loop el over: my_file {
            write el;
        }
    }
}
```

will give:

```
Polygon
```

Operators

List of operators related to obj files:

- **obj_file(string path)** : load a file (with an authorized extension) as a obj file.
- **is_obj(op)** : tests whether the operand is a OBJ file

6.6 Expressions

Expressions

—
Expressions in GAML are the value part of the [Statements161 statements]' facets. They represent or compute data that will be used as the value of the facet when the statement will be executed. An expression can be either a [literal](#) , a [unit](#) , a [constant](#) , a [ExpressionVariables161 variable], an [attribute](#) or the application of one or several [operators](#) to compose a complex expression.

6.6.1 Literals

Literals

(some literal expressions are also described in [data types](#)) A literal is a way to specify an unnamed constant value corresponding to a given data type. GAML supports various types of literals for often — or less often — used data types.

Simple Types

Values of simple (i.e. not composed) types can all be expressed using literal expressions. Namely:

- **bool** : true and false .
- **int** : decimal value, such as 100 , or hexadecimal value if preceded by '#' (e.g. #AAAAAA , which returns the int 11184810)
- **float** : the value in plain digits, using '.' for the decimal point (e.g. 123.297)
- **string** : a sequence of characters enclosed between quotes ('my string') or double quotes ("my string")

Literal Constructors

Although they are not strictly literals in the sense given above, some special constructs (called *literal constructors*) allow the modeler to declare constants of other data types. They are actually [operators](#) but can be thought of literals when used with constant operands.

- **pair** : the key and the value separated by :: (e.g. 12::'abc')
- **list** : the elements, separated by commas, enclosed inside square brackets (e.g. [12,15,15])
- **map** : a list of pairs (e.g. [12::'abc', 13::'def'])
- **point** : 2 or 3 int or float ordinates enclosed inside curly brackets (e.g. {10.0,10.0,10.0})

Universal Literal

Finally, a special literal, of type `unknown` , is shared between the data types and all the agent types (aka species). Only `bool` , `int` and `float` , which do not derive from `unknown` , do not accept this literal. All the others will accept it (e.g. `string s <- nil; is ok`).

- **unknown** : `nil` , which represents the non-initialized (or, literally, *unknown*) value.

6.6.2 Units and constants

Units and constants

 This file is automatically generated from java files. Do Not Edit It.

—
Units can be used to qualify the values of numeric variables. By default, unqualified values are considered as:

- meters for distances, lengths...
- seconds for durations
- cubic meters for volumes
- kilograms for masses

So, an expression like

```
float foo <- 1;
```

will be considered as 1 meter if foo is a distance, or 1 second if it is a duration, or 1 meter/second if it is a speed. If one wants to specify the unit, it can be done very simply by adding the unit symbol (\circ or $\#$) followed by an unit name after the numeric value, like:

```
float foo <- 1 °centimeter;
```

or

```
float foo <- 1 #centimeter;
```

In that case, the numeric value of foo will be automatically translated to 0.01 (meter). It is recommended to always use float as the type of the variables that might be qualified by units (otherwise, for example in the previous case, they might be truncated to 0). Several units names are allowed as qualifiers of numeric variables. These units represent the basic metric and US units. Composed and derived units (like velocity, acceleration, special volumes or surfaces) can be obtained by combining these units using the * and / operators. For instance:

```
float one_kmh <- 1 °km / °h const: true;
float one_millisecond <- 1 °sec / 1000;
float one_cubic_inch <- 1 °sqin * 1 °inch;
... etc ...
```

Table of Contents

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

- **#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...
 - **#pixels** (#px), value= 1.0, Comment: This unit, only available when running aspects or declaring displays, can be obtained using the same approach, but 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.
-

Length units

- **#cm** (#centimeter,#centimeters), value= 0.009999999776482582
 - **#dm** (#decimeter,#decimeters), value= 0.10000000149011612
 - **#foot** (#feet,#ft), value= 0.3047999931871891
 - **#inch** (#inches), value= 0.025399999432265757
 - **#km** (#kilometer,#kilometers), value= 1000.0
 - **#m** (#meter,#meters), value= 1.0, Comment: meter: the length basic unit
 - **#mile** (#miles), value= 1609.344
 - **#mm** (#milimeter,#milimeters), value= 9.999999776482583E-4
 - **#yard** (#yards), value= 0.9144
-

Surface units

- **#m2** , value= 1.0, Comment: square meter: the basic unit for surfaces
 - **#sqft** (#square_foot,#square_feet), value= 0.09290303584691051
 - **#sqin** (#square_inch,#square_inches), value= 6.451599711591008E-4
 - **#sqmi** (#square_mile,#square_miles), value= 2589988.110336
-

Time units

- **#day** (#days,#day), value= 86400.0
 - **#h** (#hour,#hours), value= 3600.0
 - **#minute** (#minutes,#mn), value= 60.0
 - **#month** (#months), value= 2592000.0, Comment: Note that 1 month equals 30 days and 1 year 360 days in these units
 - **#msec** (#millisecond,#milliseconds), value= 0.001
 - **#sec** (#second,#seconds,#s), value= 1.0, Comment: second: the time basic unit
 - **#year** (#years,#y), value= 3.1104E7, Comment: Note that 1 month equals 30 days and 1 year 360 days in these units
-

Volume units

- **#cl** (#centiliter,#centiliters), value= 1.0E-5
- **#dl** (#deciliter,#deciliters), value= 1.0E-4
- **#hl** (#hectoliter,#hectoliters), value= 0.1
- **#l** (#liter,#liters,#dm3), value= 0.001

- **#m3** , value= 1.0, Comment: cube meter: the basic unit for volumes
-

Weight units

- **#gram** (#grams), value= 0.001
 - **#kg** (#kilo,#kilogram,#kilos), value= 1.0, Comment: second: the basic unit for weights
 - **#ounce** (#oz,#ounces), value= 0.028349523125
 - **#pound** (#lb,#pounds,#lbm), value= 0.45359237
 - **#ton** (#tons), value= 1000.0
-

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

6.6.3 Pseudo-variables

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.

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
```

6.6.4 Variables and attributes

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](#)).

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;
}
```

6.6.5 Operators (A to K)

Operators (A to K)

~~ 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 infix notation (i.e. `operand1 operator_symbol operand1`). The ternary functional if-else operator, `? :`, uses a special infix 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 infix 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, ..., operandn)'`) or a special infix syntax with the first operand on the left-hand side of the operator name (e.g. `'operand1 operator_name(operand2, ..., operandn)'`) 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 species1 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). [Top of the page](#)

Table of Contents

Operators by categories

3D

- [box](#) , [cone3D](#) , [cube](#) , [cylinder](#) , [dem](#) , [hexagon](#) , [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](#) ,

Casting operators

- [as](#) , [as_int](#) , [as_matrix](#) , [is](#) , [is_skill](#) , [list_with](#) , [matrix_with](#) , [species](#) , [to_gaml](#) , [topology](#) ,

Color-related operators

- [-](#) , [/](#) , [^](#) , [+](#) , [blend](#) , [grayscale](#) , [hsb](#) , [rgb](#) , [rnd_color](#) ,

Comparison operators

- `!=`, `<`, `<=`, `=`, `>`, `>=`, `between`,

Containers-related operators

- `-`, `::`, `+`, `accumulate`, `among`, `at`, `collect`, `contains`, `contains_all`, `contains_any`, `count`, `empty`, `first`, `first_with`, `group_by`, `in`, `index_by`, `inter`, `interleave`, `internal_at`, `last`, `last_with`, `length`, `max`, `max_of`, `min`, `min_of`, `mul`, `one_of`, `remove_duplicates`, `reverse`, `shuffle`, `sort_by`, `sum`, `union`, `where`, `with_max_of`, `with_min_of`,

DescriptiveStatistics

- `auto_correlation`, `correlation`, `covariance`, `durbin_watson`, `kurtosis_1`, `kurtosis_2`, `moment`, `quantile`, `quantile_inverse`, `rank_interpolated`, `rms`, `skew_1`, `skew_2`, `variance1`, `variance2`,

Distributions

- `beta`, `binomial_coeff`, `binomial_complemented`, `binomial_sum`, `chi_square`, `chi_square_complemented`, `gamma`, `incomplete_beta`, `incomplete_gamma`, `incomplete_gamma_complement`, `log_gamma`, `normal_area`, `normal_density`, `normal_inverse`, `pValue_for_fStat`, `pValue_for_tStat`, `student_area`, `student_t_inverse`,

Driving operators

- `as_driving_graph`,

EDP-related operators

- `diff` , `diff2` , `internal_zero_order_equation` ,
-

Files-related operators

- `crs` , `csv_file` , `file` , `file_exists` , `folder` , `gaml_file` , `get` , `grid_file` , `image_file` , `is_csv` , `is_gaml` , `is_grid` , `is_image` , `is_obj` , `is_obj` , `is_osm` , `is_pgm` , `is_property` , `is_R` , `is_shape` , `is_svg` , `is_text` , `is_threeds` , `is_threeds` , `is_URL` , `is_xml` , `new_folder` , `obj_file` , `obj_file` , `osm_file` , `pgm_file` , `property_file` , `R_file` , `read` , `shape_file` , `svg_file` , `text_file` , `threeds_file` , `threeds_file` , `URL_file` , `writable` , `xml_file` ,
-

FIPA-related operators

- `conversation` , `message` ,
-

Graphs-related operators

- `add_edge` , `add_node` , `agent_from_geometry` , `all_pairs_shortest_path` , `alpha_index` , `as_distance_graph` , `as_edge_graph` , `as_intersection_graph` , `as_path` , `beta_index` , `betweenness_centrality` , `connected_components_of` , `connectivity_index` , `contains_edge` , `contains_vertex` , `CPU_path_between` , `degree_of` , `directed` , `edge` , `edge_between` , `edges` , `gamma_index` , `generate_barabasi_albert` , `generate_complete_graph` , `generate_watts_strogatz` , `GPU_path_between` , `grid_cells_to_graph` , `in_degree_of` , `in_edges_of` , `layout` , `load_graph_from_file` , `load_shortest_paths` , `nb_cycles` , `neighbours_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` ,
-

Iterator operators

- `accumulate`, `as_map`, `collect`, `count`, `first_with`, `frequency_of`, `group_by`, `index_by`,
`last_with`, `max_of`, `min_of`, `sort_by`, `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`, `new_predicate`,
-

Matrix-related operators

- `-`, `/`, `.`, `*`, `+`, `append_horizontally`, `append_vertically`, `column_at`, `columns_list`, `index_of`,
`last_index_of`, `row_at`, `rows_list`, `shuffle`,
-

OpenGIS

- `gml_from_wfs`, `image_from_direct_wms`, `image_from_wms`, `read_json_rest`,

Path-related operators

- `agent_from_geometry`, `all_pairs_shortest_path`, `as_path`, `CPU_path_between`,
`GPU_path_between`, `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`, `shuffle`, `truncated_gauss`,

Shape

- `antislice`, `box`, `circle`, `cone`, `cone3D`, `cube`, `cylinder`, `envelope`, `geometry_collection`,
`hemisphere`, `hexagon`, `line`, `link`, `pacman`, `plan`, `polygon`, `polyhedron`, `pyramid`,
`rectangle`, `rgbcube`, `rgbtriangle`, `slice`, `sphere`, `spherical_pie`, `square`, `teapot`, `triangle`,

Spatial operators

- `-`, `*`, `+`, `add_point`, `agent_closest_to`, `agents_at_distance`, `agents_inside`,
`agents_overlapping`, `angle_between`, `antislice`, `any_location_in`, `around`, `as_4_grid`,
`as_grid`, `as_hexagonal_grid`, `at_distance`, `at_location`, `box`, `circle`, `clean`,
`closest_points_with`, `closest_to`, `cone`, `cone3D`, `convex_hull`, `covers`, `crosses`,
`crs`, `cube`, `cylinder`, `dem`, `direction_between`, `disjoint_from`, `distance_between`,
`distance_to`, `envelope`, `farthest_point_to`, `geometry_collection`, `hemisphere`,
`hexagon`, `hierarchical_clustering`, `inside`, `inter`, `intersects`, `line`, `link`, `masked_by`,
`neighbours_at`, `neighbours_of`, `overlapping`, `overlaps`, `pacman`, `partially_overlaps`,
`path_between`, `path_to`, `plan`, `points_at`, `points_on`, `polygon`, `polyhedron`, `pyramid`,

rectangle , rgb_to_xyz , rgbcube , rgbtriangle , rotated_by , round , scaled_to , set_z , simple_clustering_by_distance , simplification , skeletonize , slice , sphere , spherical_pie , split_at , split_geometry , split_lines , square , teapot , to_rectangles , to_squares , touches , towards , transformed_by , translated_by , triangle , triangulate , union , voronoi , with_precision , without_holes ,

Spatial properties operators

- covers , crosses , intersects , partially_overlaps , touches ,
-

Spatial queries operators

- agent_closest_to , agents_at_distance , agents_inside , agents_overlapping , at_distance , closest_to , inside , neighbours_at , neighbours_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 , rotated_by , scaled_to , simplification , skeletonize , split_geometry , split_lines , 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

- `clustering_cobweb`, `clustering_DBScan`, `clustering_em`, `clustering_farthestFirst`,
`clustering_simple_kmeans`, `clustering_xmeans`, `corR`, `frequency_of`, `geometric_mean`,
`harmonic_mean`, `hierarchical_clustering`, `max`, `mean`, `mean_deviation`, `meanR`, `median`,
`min`, `mul`, `simple_clustering_by_distance`, `standard_deviation`, `sum`, `variance`,
-

Strings-related operators

- `+`, `<`, `<=`, `>`, `>=`, `as_date`, `as_time`, `at`, `char`, `contains`, `contains_all`, `contains_any`,
`copy_between`, `empty`, `first`, `in`, `index_of`, `is_number`, `last`, `last_index_of`, `length`,
`replace`, `reverse`, `sample`, `shuffle`, `split_with`,
-

System

- `.`, `copy`, `dead`, `eval_gaml`, `every`, `user_input`,
-

Time-related operators

- `as_date`, `as_time`,
-

Types-related operators

- `agent`, `bool`, `container`, `float`, `geometry`, `graph`, `int`, `list`, `map`, `matrix`, `pair`, `path`,
`string`, `unknown`,
-

User control operators

- `user_input`,
-

Water level operators

- `water_area_for` , `water_level_for` , `water_polylines_for` ,
-

Operators

- Possible use:
 - `OP(int) ---> int`
 - `OP(float) ---> float`
 - `map OP map ---> map`
 - `geometry OP container<geometry> ---> geometry`
 - `geometry OP geometry ---> geometry`
 - `container OP container ---> list`
 - `rgb OP rgb ---> rgb`
 - `matrix OP matrix ---> matrix`
 - `rgb OP int ---> rgb`
 - `int OP float ---> float`
 - `matrix OP float ---> matrix`
 - `geometry OP float ---> geometry`
 - `int OP matrix ---> matrix`
 - `float OP matrix ---> matrix`
 - `float OP float ---> float`
 - `map OP pair ---> map`
 - `int OP int ---> int`
 - `species OP agent ---> list`
 - `float OP int ---> float`
 - `point OP float ---> point`
 - `point OP int ---> point`
 - `point OP point ---> point`
 - `list OP unknown ---> list`
 - `matrix OP int ---> matrix`
- **Result:** Returns the difference of the two operands.Returns the opposite or the operand.

- **Comment:** The behavior of the operator depends on the type of the operands.
- **Special cases:**
 - 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
 - if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries
 - if the right operand is empty, - returns the left operand
 - if the right operand is an agent of the species, - returns a list containingining 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.
 - 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 var12 <- [1,2,3,4,5,6] - [2,4,9];      // var12 equals [1,3,5,6]
list var13 <- [1,2,3,4,5,6] - [0,8];      // var13 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 var14 <- rgb([255, 128, 32]) - rgb('red');      // var14 equals
rgb([0,128,32])
```

- 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 var15 <- rgb([255, 128, 32]) - 3;      // var15 equals rgb([252,125,29])
```

- if the left-hand operand is a geometry and the rigth-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) reduced by the right-hand operand distance

```
geometry var16 <- shape - 5;      // var16 equals a geometry corresponding
to the geometry of the agent applying the operator reduced by a distance of
5
```

- 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 var17 <- 3.5 - matrix([[2,5],[3,4]]);      // var17 equals
matrix([[1.5,-1.5],[0.5,-0.5]])
```

- if both operands are numbers, performs a normal arithmetic difference and returns a float if one of them is a float.

```
int var18 <- 1 - 1;      // var18 equals 0
```

- if both operands are points, returns their difference (coordinates per coordinates).

```
point var19 <- {1, 2} - {4, 5};      // var19 equals {-3.0, -3.0}
```

- if 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 var20 <- [1,2,3,4,5,6] - 2;      // var20 equals [1,3,4,5,6]
list var21 <- [1,2,3,4,5,6] - 0;      // var21 equals [1,2,3,4,5,6]
```

- **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]
int var2 <- - (-56);      // var2 equals 56
geometry var3 <- geom1 - [geom2, geom3, geom4];    // var3 equals a
geometry corresponding to geom1 - (geom2 + geom3 + geom4)
geometry var4 <- geom1 - geom2;      // var4 equals a geometry corresponding
to difference between geom1 and geom2
float var5 <- 3 - 1.2;      // var5 equals 1.8
float var6 <- 3.7 - 1.2;      // var6 equals 2.5
map var7 <- ['a'::1,'b'::2] - ('b'::2);      // var7 equals ['a'::1]
map var8 <- ['a'::1,'b'::2] - ('c'::3);      // var8 equals ['a'::1,'b'::2]
float var9 <- 1.0 - 1;      // var9 equals 0.0
point var10 <- {1, 2} - 4.5;     // var10 equals {-3.5, -2.5, -4.5}
point var11 <- {1, 2} - 4;      // var11 equals {-3.0,-2.0,-4.0}
```

- **See also:** [-](#), [+](#), [*](#), [/](#),

[Top of the page](#)

—

:

- Possible use:

- unknown OP unknown --- > unknown
- **See also:** [?](#),

[Top of the page](#)

::

- Possible use:
 - unknown OP unknown --- > 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

[Top of the page](#)

!

- Possible use:
 - OP(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](#),

[Top of the page](#)

!=

- Possible use:
 - int OP float --- > bool
 - float OP float --- > bool
 - unknown OP unknown --- > bool
 - float OP int --- > bool
- **Result:** true if both operands are different, false otherwise

- **Examples:**

```
bool var0 <- 3 != 3.0;      // var0 equals false
bool var1 <- 4 != 4.7;      // var1 equals true
bool var2 <- 3.0 != 3.0;    // var2 equals false
bool var3 <- 4.0 != 4.7;    // var3 equals true
bool var4 <- [2,3] != [2,3]; // var4 equals false
bool var5 <- [2,4] != [2,3]; // var5 equals true
bool var6 <- 3.0 != 3;      // var6 equals false
bool var7 <- 4.7 != 4;      // var7 equals true
```

- **See also:** [=](#),

[Top of the page](#)

—

?

- Possible use:
 - bool OP any expression --- > unknown
- **Result:** 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:**

```
unknown 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:** [:](#),

[Top of the page](#)

—

/

- Possible use:
 - float OP int --- > float
 - matrix OP int --- > matrix
 - int OP int --- > float
 - matrix OP float --- > matrix
 - rgb OP int --- > rgb

- matrix OP matrix --- > matrix
- rgb OP float --- > rgb
- point OP int --- > point
- float OP float --- > float
- int OP float --- > float
- point OP float --- > point
- **Result:** Returns a float, equal to the division of the left-hand operand by the right-hand operand.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 the right-hand operand is equal to zero, raises a "Division by zero" exception
 - if the right-hand operand is equal to zero, raises a "Division by zero" exception
 - 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 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 var2 <- rgb([255, 128, 32]) / 2.5;           // var2 equals rgb([102,51,13])
```

- if the left operand is a point, returns a new point with coordinates divided by the right operand

```
point var3 <- {5, 7.5} / 2.5;           // var3 equals {2, 3}
point var4 <- {2.5} / 4;           // var4 equals {0.5,1.25}
```

- **See also:** [*](#) , [+](#) , [-](#) ,

[Top of the page](#)

—

•

- Possible use:
 - matrix OP matrix --- > matrix
 - agent OP any expression --- > unknown
- **Result:** Matrix dot product.returns an evaluation of the expresion (right-hand operand) in the scope the given agent.
- **Special cases:**
 - if the agent is nil or dead, throws an exception
- **Examples:**

```
unknown var0 <- agent1.location;      // var0 equals the location of the
agent agent1
map(nil).keys
```

[Top of the page](#)

—

Λ

- Possible use:
 - int OP int --- > float
 - float OP int --- > float
 - float OP float --- > float
 - int OP float --- > 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.
- **Examples:**

```
float var0 <- 2 ^ 3;      // var0 equals 8.0
float var12 <- 4.84 ^ 0.5; // var12 equals 2.2
```

- **See also:** [*](#) , [sqrt](#) ,

[Top of the page](#)

—

(@)

Same signification as [at](#) operator.

[Top of the page](#)

—

*\`

- Possible use:
 - matrix OP int --- > matrix
 - int OP float --- > float
 - int OP matrix --- > matrix
 - point OP int --- > point
 - point OP float --- > point
 - float OP int --- > float
 - int OP int --- > int
 - matrix OP float --- > matrix
 - geometry OP point --- > geometry
 - matrix OP matrix --- > matrix
 - float OP float --- > float
 - float OP matrix --- > matrix
 - geometry OP float --- > geometry
 - rgb OP int --- > rgb
 - point OP point --- > float
- **Result:** Returns the product of the two operands.
- **Special cases:**
 - if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number
 - if both operands are numbers (float or int), performs a normal arithmetic product and returns a float if one of them is a float.
 - 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 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 var5 <- shape * {0.5,0.5,2};      // var5 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 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 var6 <- shape * 2;      // var6 equals a geometry corresponding to  
the geometry of the agent applying the operator scaled by a coefficient of  
2
```

- 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

```
rgb var7 <- rgb([255, 128, 32]) * 2;      // var7 equals rgb([255,255,64])
```

- if both operands are points, returns their scalar product

```
float var8 <- {2,5} * {4.5, 5};      // var8 equals 34.0
```

- **Examples:**

```
point var0 <- {2,5} * 4;      // var0 equals {8.0, 20.0}  
point var1 <- {2, 4} * 2.5;      // var1 equals {5.0, 10.0}  
float var2 <- 2.5 * 2;      // var2 equals 5.0  
int var3 <- 1 * 1;      // var3 equals 1
```

- **See also:** [/](#), [+](#), [-](#),

[Top of the page](#)

—

+

- Possible use:
 - matrix OP int --- > matrix
 - string OP unknown --- > string
 - container OP container --- > container
 - point OP point --- > point

- float OP matrix --- > matrix
- map OP pair --- > map
- matrix OP matrix --- > matrix
- geometry OP float --- > geometry
- float OP int --- > float
- int OP matrix --- > matrix
- int OP int --- > int
- float OP float --- > float
- matrix OP float --- > matrix
- geometry OP geometry --- > geometry
- container OP unknown --- > list
- string OP string --- > string
- rgb OP rgb --- > rgb
- geometry OP map --- > geometry
- point OP float --- > point
- int OP float --- > float
- rgb OP int --- > rgb
- map OP map --- > map
- point OP int --- > point
- **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 the right-operand is a point, a geometry or an agent, returns the geometry resulting from the union between both geometries
 - if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one being casted into a string)
 - if the left-hand operand is a geometry and the right-hand operand a map (with [distance::float, quadrantSegments:: int (the number of line segments used to represent a quadrant of a circle), endCapStyle::int (1: (default) a semi-circle, 2: a straight line perpendicular to the end segment, 3: a half-square)]), returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged considering the right-hand operand parameters
 - if the left-operand is a string, concatenates both operands;

```
string var0 <- "hello " + 12;      // var0 equals "hello 12"
```

- if both operands are points, returns their sum.

```
point var1 <- {1, 2} + {4, 5};      // var1 equals {5.0, 7.0}
```

- 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

```
geometry var2 <- shape + 5;      // var2 equals a geometry corresponding to
the geometry of the agent applying the operator enlarged by a distance of 5
```

- 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 var3 <- 3.5 + matrix([[2,5],[3,4]]);      // var3 equals
matrix([[5.5,8.5],[6.5,7.5]])
```

- 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 var4 <- 1 + 1;      // var4 equals 2
```

- 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 var5 <- [1,2,3,4,5,6] + 2;      // var5 equals [1,2,3,4,5,6,2]
list var6 <- [1,2,3,4,5,6] + 0;      // var6 equals [1,2,3,4,5,6,0]
```

- if both operands are colors, returns a new color resulting from the sum of the two operands, component by component

```
rgb var7 <- rgb([255, 128, 32]) + rgb('red');      // var7 equals
rgb([255,128,32])
```

- if 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 var8 <- {1, 2} + 4;      // var8 equals {5.0, 6.0,4.0}
point var9 <- {1, 2} + 4.5;     // var9 equals {5.5, 6.5,4.5}
```

- 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 var10 <- rgb([255, 128, 32]) + 3;      // var10 equals rgb([255,131,35])
```

- **Examples:**

```

container var11 <- [1,2,3,4,5,6] + [2,4,9];      // var11 equals
[1,2,3,4,5,6,2,4,9]
container var12 <- [1,2,3,4,5,6] + [0,8];      // var12 equals
[1,2,3,4,5,6,0,8]
map var13 <- ['a'::1,'b'::2] + ('c'::3);      // var13 equals
['a'::1,'b'::2,'c'::3]
map var14 <- ['a'::1,'b'::2] + ('c'::3);      // var14 equals
['a'::1,'b'::2,'c'::3]
float var15 <- 1.0 + 1;      // var15 equals 2.0
float var16 <- 1.0 + 2.5;     // var16 equals 3.5
geometry var17 <- geom1 + geom2;      // var17 equals a geometry
corresponding to union between geom1 and geom2
geometry var18 <- shape + ["distance":5.0, "quadrantSegments":4,
"endCapStyle": 2];      // var18 equals a geometry corresponding to the
geometry of the agent applying the operator enlarged by a distance of 5,
with 4 segments to represent a quadrant of a circle and a straight line
perpendicular to the end segment
float var19 <- 2 + 2.5;      // var19 equals 4.5
map var20 <- ['a'::1,'b'::2] + ['c'::3];      // var20 equals
['a'::1,'b'::2,'c'::3]
map var21 <- ['a'::1,'b'::2] + [5::3.0];      // var21 equals
['a'::1.0,'b'::2.0,5::3.0]

```

- **See also:** [-](#), [/](#), [*](#),

[Top of the page](#)

—

<

- Possible use:
 - int OP int --- > bool
 - float OP int --- > bool
 - int OP float --- > bool
 - float OP float --- > bool
 - point OP point --- > bool
 - string OP string --- > 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 points, returns true if only if left component (x) of the left operand is 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 var4 <- {5,7} < {4,6};      // var4 equals false
bool var5 <- {5,7} < {4,8};      // var5 equals false

```

- if both operands are String, uses a lexicographic comparison of two strings

```
bool var6 <- 'abc' < 'aeb';      // var6 equals true
```

- **Examples:**

```
bool var0 <- 3 < 7;      // var0 equals true
bool var1 <- 3.5 < 7;      // var1 equals true
bool var2 <- 3 < 2.5;      // var2 equals false
bool var3 <- 3.5 < 7.6;      // var3 equals true
```

[Top of the page](#)

—

<=

- Possible use:
 - float OP int --- > bool
 - string OP string --- > bool
 - int OP float --- > bool
 - int OP int --- > bool
 - float OP float --- > bool
 - point OP point --- > 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 var4 <- 'abc' <= 'aeb';      // var4 equals true
```

- if both operands are points, returns true if only if left component (x) of the left operand is 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 <- 7.0 <= 7;      // var0 equals true
```

```
bool var1 <- 3 <= 2.5;      // var1 equals false
bool var2 <- 3 <= 7;        // var2 equals true
bool var3 <- 3.5 <= 3.5;    // var3 equals true
```

[Top of the page](#)

<>

Same signification as [!=](#) operator.

[Top of the page](#)

'=='

- Possible use:
 - int OP float --- > bool
 - float OP int --- > bool
 - unknown OP unknown --- > bool
 - int OP int --- > bool
 - float OP float --- > bool
- **Result:** returns true if both operands are equal, false otherwise
- **Examples:**

```
bool var0 <- 3 = 3.0;      // var0 equals true
bool var1 <- 4 = 4.7;      // var1 equals false
bool var2 <- 3.0 = 3;      // var2 equals true
bool var3 <- 4.7 = 4;      // var3 equals false
bool var4 <- 3.0 = 3;      // var4 equals true
bool var5 <- [2,3] = [2,3]; // var5 equals true
bool var6 <- 3 = 3;        // var6 equals true
bool var7 <- 4 = 5;        // var7 equals false
bool var8 <- 3 = 3;        // var8 equals true
bool var9 <- 4.5 = 4.7;    // var9 equals false
```

- **See also:** [!=](#),

[Top of the page](#)

>

- Possible use:
 - string OP string --- > bool
 - int OP float --- > bool
 - float OP float --- > bool
 - point OP point --- > bool
 - float OP int --- > bool
 - int OP 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 String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' > 'aeb';      // var0 equals false
```

- if both operands are points, returns true if only if left component (x) of the left operand is 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 var1 <- {5,7} > {4,6};      // var1 equals true
bool var2 <- {5,7} > {4,8};      // var2 equals false
```

- **Examples:**

```
bool var3 <- 3 > 2.5;      // var3 equals true
bool var4 <- 3.5 > 7.6;      // var4 equals false
bool var5 <- 3.5 > 7;      // var5 equals false
bool var6 <- 3 > 7;      // var6 equals false
```

[Top of the page](#)

—

>=

- Possible use:
 - int OP float --- > bool
 - float OP int --- > bool
 - float OP float --- > bool
 - int OP int --- > bool
 - string OP string --- > bool
 - point OP point --- > 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 String, uses a lexicographic comparison of two strings

```
bool var4 <- 'abc' >= 'aeb';      // var4 equals false
bool var5 <- 'abc' >= 'abc';      // var5 equals true
```

- if both operands are points, returns true if only if left component (x) of the left operand is greater 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 var6 <- {5,7} >= {4,6};      // var6 equals true
bool var7 <- {5,7} >= {4,8};      // var7 equals false
```

- **Examples:**

```
bool var0 <- 3 >= 2.5;      // var0 equals true
bool var1 <- 3.5 >= 7;      // var1 equals false
bool var2 <- 3.5 >= 3.5;      // var2 equals true
bool var3 <- 3 >= 7;      // var3 equals false
```

[Top of the page](#)

abs

- Possible use:
 - OP(float) ---> float
 - OP(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
```

[Top of the page](#)

accumulate

- Possible use:
 - container OP any expression --- > list
- **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 right-hand operand elements.
- **Examples:**

```
list var0 <- [a1,a2,a3] accumulate (each neighbours_at 10);      // var0  
equals a flat list of all the neighbours of these three agents  
list var1 <- [1,2,4] accumulate ([2,4]);      // var1 equals [2,4,2,4,2,4]
```

- **See also:** [collect](#),

[Top of the page](#)

acos

- Possible use:
 - OP(int) --- > float
 - OP(float) --- > 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](#) ,

[Top of the page](#)

add_edge

- Possible use:
 - graph OP pair --- > graph
- **Result:** add an edge between source vertex and the target vertex

- **Comment:** If the edge already exists the graph is unchanged
- **Examples:**

```
graph <- graph add_edge (source::target);
```

- **See also:** [G OperatorsLZ#],

[Top of the page](#)

add_node

- Possible use:
 - graph OP geometry ---> graph
- **Result:** adds a node in a graph.
- **Examples:**

```
graph var0 <- graph add_node node(0) ;           // var0 equals the graph with
node(0)
```

[Top of the page](#)

add_point

- Possible use:
 - geometry OP point ---> geometry
- **Result:** A geometry resulting from the addition of the right point (coordinate) to the left-hand geometry
- **Examples:**

```
geometry var0 <- square(5) add_point {10,10} ;           // var0 equals a pentagon
```

[Top of the page](#)

agent

- Possible use:
 - OP(any) ---> agent

- **Result:** Casts the operand into the type agent

[Top of the page](#)

agent_closest_to

- Possible use:
 - OP(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:** [neighbours_at](#) , [neighbours_of](#) , [agents_inside](#) , [agents_overlapping](#) , [closest_to](#) , [inside](#) , [overlapping](#) ,

[Top of the page](#)

agent_from_geometry

- Possible use:
 - path OP 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);
```

[Top of the page](#)

agents_at_distance

- Possible use:
 - OP(float) --- > list
- **Result:** A list of agents situated at a distance \leq the right argument.
- **Comment:** Equivalent to neighbours_at with a left-hand argument equal to 'self'
- **Examples:**

```
list var0 <- agents_at_distance(20);      // var0 equals all the agents
(excluding the caller) which distance to the caller is  $\leq$  20
```

- **See also:** [neighbours_at](#) , [neighbours_of](#) , [agent_closest_to](#) , [agents_inside](#) , [closest_to](#) , [inside](#) , [overlapping](#) , [at_distance](#) ,

[Top of the page](#)

agents_inside

- Possible use:
 - OP(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 return 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](#) ,

[Top of the page](#)

agents_overlapping

- Possible use:
 - OP(unknown) --- > list<agent>
- **Result:** A list of agents overlapping the operand (casted as a geometry).
- **Examples:**

```
list<agent> var0 <- agents_overlapping(self);      // var0 equals return the
agents that overlap the shape of the agent applying the operator.
```

- **See also:** [neighbours_at](#) , [neighbours_of](#) , [agent_closest_to](#) , [agents_inside](#) , [closest_to](#) , [inside](#) , [overlapping](#) , [at_distance](#) ,

[Top of the page](#)

—

all_pairs_shortest_path

- Possible use:
 - `OP(graph) ---> matrix`
- **Result:** return a matrix containing all pairs of shortest paths (rows: source, columns: target)
- **Examples:**

```
matrix var0 <- mall_pairs_shortest_paths(my_graph);           // var0 equals
shortest_paths_matrix will contain all pairs of shortest paths
```

[Top of the page](#)

—

alpha_index

- Possible use:
 - `OP(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. $\text{alpha} = \text{nb_cycles} / (2 * S - 5)$ - 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](#) ,

[Top of the page](#)

—

among

- Possible use:
 - `int OP container ---> list`

- **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. If it is smaller or equal to zero, it returns an empty list
- **Examples:**

```
list var0 <- 3 among [1,2,4,3,5,7,6,8];      // var0 equals [1,2,8] (for example)
list var1 <- 3 among g2;          // var1 equals [node6,node11,node7]
list var2 <- 3 among list(node);    // var2 equals [node1,node11,node4]
```

[Top of the page](#)

and

- Possible use:
 - bool OP 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](#) ,

[Top of the page](#)

angle_between

- Possible use:
 - point OP point --- > int
- **Result:** the angle between vector [P0P1] and [P0P2]
- **Examples:**

```
int var0 <- angle_between({5,5},{10,5},{5,10});      // var0 equals 90
```

[Top of the page](#)

antislice

- Possible use:
 - float OP float --- > geometry
- **Result:** A sphere geometry which radius is equal to the operand made of 2 hemispheres.
- **Comment:** the centre of the sphere is by default the location of the current agent that is calling this operator.
- **Special cases:**
 - returns a point if the operand is lower or equal to 0.
- **Examples:**

```
geometry var0 <- antislice(10,0.3);      // var0 equals a circle geometry of
radius 10, displayed as an antislice.
```

- **See also:** [around](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#) , [hemisphere](#) , [pie3D](#) ,

[Top of the page](#)

any

Same signification as [one_of](#) operator.

[Top of the page](#)

any_location_in

- Possible use:
 - OP(geometry) --- > point
- **Result:** A point inside (or touching) the operand-geometry.
- **Examples:**

```
point var0 <- any_location_in(square(5));      // var0 equals a point of the
square, for example : {3,4.6}.
```

- **See also:** [closest_points_with](#) , [farthest_point_to](#) , [points_at](#) ,

[Top of the page](#)

any_point_in

Same signification as [any_location_in](#) operator.

[Top of the page](#)

append_horizontally

- Possible use:
 - matrix OP matrix --- > matrix
 - matrix OP 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]])
```

[Top of the page](#)

append_vertically

- Possible use:
 - matrix OP matrix --- > matrix
 - matrix OP 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]])
```

[Top of the page](#)

around

- Possible use:
 - float OP 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 a the ring
geometry between 5 and 10.
```

- **See also:** [circle](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#)

[Top of the page](#)

as

- Possible use:
 - unknown OP any expression --- > unknown
- **Result:** casting of the argument into a given type

[Top of the page](#)

as_4_grid

- Possible use:
 - geometry OP point --- > matrix
- **Result:** A matrix of square geometries (grid with 4-neighbourhood) 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 matrix of square
geometries (grid with 4-neighbourhood) with 10 columns and 5 lines
corresponding to the square tessellation of the geometry of the agent
applying the operator.
```

[Top of the page](#)

as_date

- Possible use:
 - OP(float) --- > string
 - float OP string --- > string
- **Result:** converts a number into a string with year, month, day, hour, minutes, second following a given pattern (right-hand operand)
- **Special cases:**
 - used as an unary operator, 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](#) ,

[Top of the page](#)

as_distance_graph

- Possible use:
 - container OP float --- > graph
 - container OP map --- > graph
 - container OP float --- > 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.as_distance_graph is more efficient for a list of points than as_intersection_graph.as_distance_graph is more efficient for a list of points than as_intersection_graph.
- **Examples:**

```
list(ant) as_distance_graph 3.0;
```

```
list(ant) as_distance_graph 3.0
list(ant) as_distance_graph 3.0;
```

- **See also:** [as_intersection_graph](#) , [as_edge_graph](#) ,

[Top of the page](#)

—

as_driving_graph

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

—

as_edge_graph

- Possible use:
 - OP(container) --- > graph
 - OP(map) --- > graph
- **Result:** creates a graph from the list/map of edges given as operand
- **Special cases:**
 - if the operand is a list, the graph will be built with elements of the list as vertices

```
graph var0 <- as_edge_graph([{1,5},{12,45},{34,56}]);           // var0 equals
build a graph with these three vertices and reflexive links on each
vertices
```

- if the operand is a map, the graph will be built by creating edges from pairs of the map

```
graph var1 <- as_edge_graph([{1,5}:{12,45},{12,45}:{34,56}]);      // var1
equals a graph with these three vertices and two edges
```

- **See also:** [as_intersection_graph](#) , [as_distance_graph](#) ,

[Top of the page](#)

as_grid

- Possible use:
 - geometry OP point --- > matrix
- **Result:** A matrix of square geometries (grid with 8-neighbourhood) 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-neighbourhood) with 10 columns and 5 lines
corresponding to the square tessellation of the geometry of the agent
applying the operator.
```

[Top of the page](#)

as_hexagonal_grid

- Possible use:
 - geometry OP point --- > list<geometry>
- **Result:** A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand list of geometries
- **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.
```

[Top of the page](#)

as_int

- Possible use:
 - string OP 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](#),

[Top of the page](#)

as_intersection_graph

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

as_map

- Possible use:

- container OP 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 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 var1 <- [1::2,3::4,5::6] as_map (each::(each * 2));      // var1 equals
[2::4, 4::8, 6::12]
```

[Top of the page](#)

as_matrix

- Possible use:
 - unknown OP point --- > matrix
- **Result:** casts the left operand into a matrix with right operand as preferenced 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 too 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](#),

[Top of the page](#)

as_path

- Possible use:
 - list<geometry> OP 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
```

[Top of the page](#)

as_time

- Possible use:
 - OP(float) --- > string
- **Result:** converts the given number into a string with hours, minutes and seconds
- **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](#),

[Top of the page](#)

asin

- Possible use:
 - OP(int) --- > float
 - OP(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](#) ,

[Top of the page](#)

at

- Possible use:

- `msi.gama.util.IContainer <[KeyType] , [ValueType]> .Addressable <[KeyType] , [ValueType]> OP [KeyType] ---> [ValueType]`
- `string OP int ---> string`
- **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 `[IndexOutOfBoundsException]`. 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 var0 <- [1, 2, 3] at 2;      // var0 equals 3
point var1 <- [{1,2}, {3,4}, {5,6}] at 0;      // var1 equals {1.0,2.0}
```

- **Examples:**

```
string var2 <- 'abcdef' at 0;      // var2 equals 'a'
```

- **See also:** [contains_any](#) [contains_all](#), [contains_any](#),

[Top of the page](#)

at_distance

- Possible use:
 - `msi.gama.util.IContainer < ?,? extends msi.gama.metamodel.shape.IShape> OP float ---> list<agent>`
- **Result:** A list of agents among the left-operand list that are located at a distance \leq the right operand from the caller agent (in its topology)
- **Examples:**

```
list<agent> var0 <- [ag1, ag2, ag3] at_distance 20;      // var0 equals the
agents of the list located at a distance  $\leq$  20 from the caller agent (in
the same order).
```

- **See also:** [neighbours_at](#) , [neighbours_of](#) , [agent_closest_to](#) , [agents_inside](#) , [closest_to](#) , [inside](#) , [overlapping](#) ,

[Top of the page](#)

—

at_location

- Possible use:
 - geometry OP 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).
```

[Top of the page](#)

—

atan

- Possible use:
 - OP(float) --- > float
 - OP(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](#) ,

[Top of the page](#)

—

atan2

- Possible use:
 - float OP 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](#) ,

[Top of the page](#)

auto_correlation

- Possible use:
 - container OP int --- > float
- **Result:** The auto-correlation

[Top of the page](#)

beta

- Possible use:
 - float OP float --- > float

[Top of the page](#)

beta_index

- Possible use:
 - OP(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) : $\text{beta} = e/v$.
- **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](#) ,

[Top of the page](#)

between

- Possible use:
 - int OP int --- > bool
 - float OP float --- > bool
- **Result:** returns true if the first integer operand is bigger than the second integer operand and smaller than the third integer operand; returns true if the first float operand is bigger than the second float operand and smaller than the third float operand
- **Examples:**

```
bool var0 <- between(5, 1, 10);      // var0 equals true  
bool var1 <- between(5.0, 1.0, 10.0);      // var1 equals true
```

[Top of the page](#)

betweenness_centrality

- Possible use:
 - OP(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
```

[Top of the page](#)

binomial

- Possible use:
 - OP(point) --- > int
- **Result:** A value from a random variable following a binomial distribution. The operand {n,p} represents 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](#) ,

[Top of the page](#)

binomial_coeff

- Possible use:
 - int OP int --- > float

[Top of the page](#)

binomial_complemented

- Possible use:
 - int OP int --- > float

[Top of the page](#)

binomial_sum

- Possible use:
 - int OP int --- > float

[Top of the page](#)

blend

- Possible use:
 - `rgb OP rgb --- > rgb`
 - `rgb OP rgb --- > 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](#) ,

[Top of the page](#)

bool

- Possible use:
 - `OP(any) --- > bool`
- **Result:** Casts the operand into the type bool

[Top of the page](#)

box

- Possible use:
 - `OP(point) --- > geometry`
 - `float OP float --- > geometry`
- **Result:** A box geometry which side sizes are given by the operands.
- **Comment:** the centre of the box is by default the location of the current agent in which has been called this operator.the centre 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](#) ,

[Top of the page](#)

buffer

Same signification as [+](#) operator.

[Top of the page](#)

ceil

- Possible use:

- OP(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](#) ,

[Top of the page](#)

char

- Possible use:
 - OP(int) --- > string
- **Special cases:**
 - converts ASCII integer value to character

```
string var0 <- char (34);      // var0 equals '''
```

[Top of the page](#)

chi_square

- Possible use:
 - float OP float --- > float

[Top of the page](#)

chi_square_complemented

- Possible use:
 - float OP float --- > float

[Top of the page](#)

circle

- Possible use:
 - OP(float) --- > geometry
- **Result:** A circle geometry which radius is equal to the operand.
- **Comment:** the centre 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.
- **Examples:**

```
geometry var0 <- circle(10);      // var0 equals a geometry as a circle of
radius 10.
```

- **See also:** [around](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#) ,

[Top of the page](#)

clean

- Possible use:
 - OP(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.
```

[Top of the page](#)

closest_points_with

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

closest_to

- Possible use:

- `msi.gama.util.IContainer < ?,? extends msi.gama.metamodel.shape.IShape > OP unknown ---> agent`
- **Result:** An agent 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:**

```
agent 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:** [neighbours_at](#) , [neighbours_of](#) , [inside](#) , [overlapping](#) , [agents_overlapping](#) , [agents_inside](#) , [agent_closest_to](#) ,

[Top of the page](#)

clustering_cobweb

- Possible use:
 - `??? OP msi.gama.util.IList < java.lang.String > ---> list<list<agent>>`
- **Result:** A list of agent groups clustered by [CobWeb] Algorithm based on the given attributes. Some parameters can be defined: acuity: minimum standard deviation for numeric attributes; cutoff: category utility threshold by which to prune nodes seed
- **Examples:**

```
list<list<agent>> var0 <- clustering_cobweb([ag1, ag2, ag3, ag4, ag5],
["size", "age", "weight"], ["acuity":3.0, "cutoff":0.5]);      // var0 equals
for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_xmeans](#) , [clustering_em](#) , [clustering_farthestFirst](#) , [clustering_simple_kmeans](#) , [clustering_cobweb](#) ,

[Top of the page](#)

clustering_DBScan

- Possible use:
 - `??? OP msi.gama.util.IList < java.lang.String > ---> list<list<agent>>`

- **Result:** A list of agent groups clustered by DBScan Algorithm based on the given attributes. Some parameters can be defined: distance_f: The distance function to use for instances comparison (euclidean or manhattan); min_points: minimum number of [DataObjects] required in an epsilon-range-queryepsilon: epsilon -- radius of the epsilon-range-queries
- **Examples:**

```
list<list<agent>> var0 <- clustering_DBScan([ag1, ag2, ag3, ag4, ag5],  
["size", "age", "weight"], ["distance_f"::"manhattan"]); // var0 equals  
for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_xmeans](#) , [clustering_em](#) , [clustering_farthestFirst](#) , [clustering_simple_kmeans](#) , [clustering_cobweb](#) ,

[Top of the page](#)

clustering_em

- Possible use:
 - ??? OP msi.gama.util.IList < java.lang.String > --- > list<list<agent>>
- **Result:** A list of agent groups clustered by EM Algorithm based on the given attributes. Some parameters can be defined: max_iterations: the maximum number of iterations to perform;num_clusters: the number of clusters; minStdDev: minimum allowable standard deviation
- **Examples:**

```
list<list<agent>> var0 <- clustering_em([ag1, ag2, ag3, ag4, ag5],  
["size", "age", "weight"], ["max_iterations":10, "num_clusters":3]); //  
var0 equals for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_xmeans](#) , [clustering_simple_kmeans](#) , [clustering_farthestFirst](#) , [clustering_DBScan](#) , [clustering_cobweb](#) ,

[Top of the page](#)

clustering_farthestFirst

- Possible use:
 - ??? OP msi.gama.util.IList < java.lang.String > --- > list<list<agent>>
- **Result:** A list of agent groups clustered by Farthest First Algorithm based on the given attributes. Some parameters can be defined: num_clusters: the number of clusters
- **Examples:**

```
list<list<agent>> var0 <- clustering_farthestFirst([ag1, ag2, ag3, ag4, ag5], ["size", "age", "weight"], ["num_clusters":3]); // var0 equals for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_xmeans](#) , [clustering_simple_kmeans](#) , [clustering_em](#) , [clustering_DBScan](#) , [clustering_cobweb](#) ,

[Top of the page](#)

clustering_simple_kmeans

- Possible use:
 - ??? OP msi.gama.util.IList < java.lang.String > --- > list<list<agent>>
- **Result:** A list of agent groups clustered by K-Means Algorithm based on the given attributes. Some parameters can be defined: distance_f: The distance function to use. 4 possible distance functions: euclidean (by default) ; 'chebyshev', 'manhattan' or 'levenshtein'; dont_replace_missing_values: if false, replace missing values globally with mean/mode; max_iterations: the maximum number of iterations to perform; num_clusters: the number of clusters
- **Examples:**

```
list<list<agent>> var0 <- clustering_simple_kmeans([ag1, ag2, ag3, ag4, ag5], ["size", "age", "weight"], ["distance_f": "manhattan", "num_clusters":3]); // var0 equals for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_xmeans](#) , [clustering_em](#) , [clustering_farthestFirst](#) , [clustering_DBScan](#) , [clustering_cobweb](#) ,

[Top of the page](#)

clustering_xmeans

- Possible use:
 - ??? OP msi.gama.util.IList < java.lang.String > --- > list<list<agent>>
- **Result:** A list of agent groups clustered by X-Means Algorithm based on the given attributes. Some parameters can be defined: bin_value: value given for true value of boolean attributes; cut_off_factor: the cut-off factor to use; distance_f: The distance function to use. 4 possible distance functions: euclidean (by default) ; 'chebyshev', 'manhattan' or 'levenshtein'; max_iterations: the maximum number of iterations to perform; max_kmeans: the maximum number of iterations to perform in KMeans; max_kmeans_for_children: the maximum number

of iterations KMeans that is performed on the child centers; max_num_clusters: the maximum number of clusters; min_num_clusters: the minimal number of clusters

- **Examples:**

```
list<list<agent>> var0 <- clustering_xmeans([ag1, ag2, ag3,
ag4, ag5],["size", "age", "weight", "is_male"], ["bin_value"]::1.0,
"distance_f"::"manhattan", "max_num_clusters"::10, "min_num_clusters"::2]);
// var0 equals for example, can return [[ag1, ag3], [ag2], [ag4, ag5]]
```

- **See also:** [clustering_simple_kmeans](#) , [clustering_em](#) , [clustering_farthestFirst](#) , [clustering_DBScan](#) , [clustering_cobweb](#) ,

[Top of the page](#)

collect

- Possible use:
 - container OP any expression --- > list
- **Result:** returns a new list, in which each element is the evaluation of the right-hand operand.
- **Comment:** collect is very similar to accumulate except. Nevertheless if the evaluation of the right-hand operand produces a list, the returned list is a list of list of elements. In contrarily, the list produced by accumulate is only a list of elements (all the lists) produced are concatenated. In addition, collect can be applied to any container.
- **Special cases:**
 - if the left-hand operand is nil, collect throws an error
- **Examples:**

```
list var0 <- [1,2,4] collect (each *2);      // var0 equals [2,4,8]
list var1 <- [1,2,4] collect ([2,4]);        // var1 equals [[2,4],[2,4],
[2,4]]
list var2 <- [1::2, 3::4, 5::6] collect (each + 2);    // var2 equals
[4,6,8]
list var3 <- (list(node) collect (node(each).location.x * 2)); // var3
equals the list of nodes with their x multiplied by 2
```

- **See also:** [accumulate](#) ,

[Top of the page](#)

column_at

- Possible use:
 - matrix OP 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](#) ,

[Top of the page](#)

columns_list

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

cone

- Possible use:
 - OP(point) --- > geometry
- **Result:** A cone geometry which min and max angles are given by the operands.
- **Comment:** the centre 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.
- **Examples:**

```
geometry var0 <- cone({0, 45});      // var0 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](#) ,

[Top of the page](#)

cone3D

- Possible use:
 - float OP float --- > geometry
- **Result:** A cone geometry which radius is equal to the operand.
- **Comment:** the centre 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](#) ,

[Top of the page](#)

connected_components_of

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

connectivity_index

- Possible use:
 - OP(graph) --- > float
- **Result:** retruns a simple connetivity 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](#) ,

[Top of the page](#)

container

- Possible use:
 - OP(any) --- > container
- **Result:** Casts the operand into the type container

[Top of the page](#)

contains

- Possible use:
 - container OP unknown --- > bool
 - string OP string --- > 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 it is a map, contains returns true if the operand is a key of the map
 - if it is a file, contains returns true if 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 both operands are strings, returns true if the right-hand operand contains the right-hand pattern;
- if it is a list or a matrix, contains returns true if the list or matrix contains the right operand

```
bool var0 <- [1, 2, 3] contains 2;      // var0 equals true
bool var1 <- [{1,2}, {3,4}, {5,6}] contains {3,4}; // var1 equals true
```

- **Examples:**

```
bool var2 <- 'abcded' contains 'bc'; // var2 equals true
```

- **See also:** [contains_any](#) [contains_all](#), [contains_any](#),

[Top of the page](#)

contains_all

- Possible use:
 - container OP container --- > bool
 - string OP 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](#) ,

[Top of the page](#)

contains_any

- Possible use:
 - container OP container --- > bool
 - string OP list --- > 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 <- [1,2,3,4,5,6] contains_any [2,4];      // var0 equals true
bool var1 <- [1,2,3,4,5,6] contains_any [2,8];      // var1 equals true
bool var2 <- [1::2, 3::4, 5::6] contains_any [1,3];    // var2 equals
false
bool var3 <- [1::2, 3::4, 5::6] contains_any [2,4];    // var3 equals true
bool var4 <- "abcabca" contains_any ["ca","xy"];       // var4 equals true
```

- **See also:** [contains](#) , [contains_all](#) ,

[Top of the page](#)

contains_edge

- Possible use:
 - graph OP unknown --- > bool
 - graph OP pair --- > bool
- **Result:** returns true if the graph(left-hand operand) contains the given edge (right-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](#),

[Top of the page](#)

contains_vertex

- Possible use:
 - graph OP unknown --- > bool
- **Result:** returns true if the graph(left-hand operand) contains the given vertex (right-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](#),

[Top of the page](#)

conversation

- Possible use:
 - OP(unknown) --- > msi.gaml.extensions.fipa.Conversation

[Top of the page](#)

convex_hull

- Possible use:
 - OP(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
```

[Top of the page](#)

copy

- Possible use:
 - OP(unknown) ---> unknown
- **Result:** returns a copy of the operand.

[Top of the page](#)

copy_between

- Possible use:
 - string OP int ---> string
 - list OP int ---> list
- **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"  
list var1 <- copy_between ([4, 1, 6, 9 ,7], 1, 3);    // var1 equals [1,  
6]
```

[Top of the page](#)

corR

- Possible use:
 - container OP 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
```

[Top of the page](#)

correlation

- Possible use:
 - container OP container --- > float

[Top of the page](#)

COS

- Possible use:
 - OP(int) --- > float
 - OP(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.
- **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](#) ,

[Top of the page](#)

cos_rad

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

Count

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

covariance

- Possible use:
 - container OP container --- > float

[Top of the page](#)

COVERS

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

CPU_path_between

- Possible use:
 - graph OP geometry --- > path
- **Result:** The shortest path between a list of two objects in a graph computed with CPU
- **Examples:**

```
path var0 <- my_graph CPU_path_between (ag1:: ag2);           // var0 equals A
path between ag1 and ag2
```

[Top of the page](#)

CROSSES

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

crs

- Possible use:
 - OP(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
```

[Top of the page](#)

csv_file

- Possible use:
 - OP(string) ---> file
- **Result:** Constructs a file of type csv. Allowed extensions are limited to csv, tsv

[Top of the page](#)

cube

- Possible use:
 - OP(float) ---> geometry

- **Result:** A cube geometry which side size is equal to the operand.
- **Comment:** the centre 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](#) ,

[Top of the page](#)

cylinder

- Possible use:
 - float OP float --- > geometry
- **Result:** A cylinder geometry which radius is equal to the operand.
- **Comment:** the centre 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](#) ,

[Top of the page](#)

dead

- Possible use:
 - OP(agent) --- > bool
- **Result:** true if the agent is dead, false otherwise.
- **Examples:**

```
bool var0 <- dead(agent_A);      // var0 equals true or false
```

[Top of the page](#)

degree_of

- Possible use:
 - graph OP 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](#) ,

[Top of the page](#)

dem

- Possible use:
 - OP(file) --- > geometry
 - file OP file --- > geometry
 - file OP float --- > geometry
 - file OP file --- > geometry
- **Result:** A polygon that is equivalent to the surface of the texture
- **Special cases:**
 - a point if the operand is lower or equal to 0.
- **Examples:**

```
geometry var0 <- dem(dem);      // var0 equals returns a geometry as a
rectangle of weight and height equal to the texture.
geometry var1 <- dem(dem,texture);      // var1 equals a geometry as a
rectangle of weight and height equal to the texture.
geometry var2 <- dem(dem,texture,z_factor);      // var2 equals a geometry
as a rectangle of weight and height equal to the texture.
geometry var3 <- dem(dem,z_factor);      // var3 equals a geometry as a
rectangle of weight and height equal to the texture.
```

[Top of the page](#)

diff

- Possible use:
 - float OP float --- > float

[Top of the page](#)

diff2

- Possible use:
 - float OP float --- > float

[Top of the page](#)

directed

- Possible use:
 - OP(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](#),

[Top of the page](#)

direction_between

- Possible use:
 - topology OP 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](#) ,

[Top of the page](#)

direction_to

Same signification as [towards](#) operator.

[Top of the page](#)

disjoint_from

- Possible use:
 - geometry OP geometry --- > bool
- **Result:** A boolean, equal to true if the left-geometry (or agent/point) is disjoint 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](#) ,

[Top of the page](#)

distance_between

- Possible use:

- topology OP 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](#) ,

[Top of the page](#)

distance_to

- Possible use:
 - geometry OP geometry --- > float
 - point OP point --- > 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](#) ,

[Top of the page](#)

div

- Possible use:
 - float OP float --- > int
 - int OP int --- > int
 - float OP int --- > int
 - int OP float --- > 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 <- 40.1 div 4.5;      // var0 equals 8
int var1 <- 40 div 3;        // var1 equals 13
int var2 <- 40.5 div 3;      // var2 equals 13
int var3 <- 40 div 4.1;      // var3 equals 9
```

- **See also:** [mod](#),

[Top of the page](#)

—

durbin_watson

- Possible use:
 - OP(container) --- > float

[Top of the page](#)

—

edge

- Possible use:
 - OP(unknown) --- > unknown
 - OP(pair) --- > unknown
 - unknown OP float --- > unknown
 - unknown OP unknown --- > unknown
 - pair OP float --- > unknown
 - unknown OP unknown --- > unknown
 - unknown OP unknown --- > unknown
 - pair OP unknown --- > unknown
 - unknown OP unknown --- > unknown

[Top of the page](#)

—

edge_between

- Possible use:
 - graph OP 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](#),

[Top of the page](#)

edges

- Possible use:
 - OP(container) ---> container

[Top of the page](#)

empty

- Possible use:
 - OP(container) ---> bool
 - OP(string) ---> 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 list, empty returns true if there is no element in the list, and false otherwise

```
bool var0 <- empty([]);      // var0 equals true
```

- if it is a string, empty returns true if the string does not contain any character, and false otherwise

```
bool var1 <- empty ('abcd');      // var1 equals false
```

[Top of the page](#)

enlarged_by

Same signification as [+ operator](#).

[Top of the page](#)

envelope

- Possible use:
 - OP(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

[Top of the page](#)

eval_gaml

- Possible use:
 - OP(string) ---> unknown
- **Result:** evaluates the given GAML string.
- **Examples:**

```
unknown var0 <- eval_gaml("2+3");      // var0 equals 5
```

[Top of the page](#)

even

- Possible use:
 - OP(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
```

[Top of the page](#)

every

- Possible use:
 - OP(int) --- > bool
- **Result:** true every operand time step, false otherwise
- **Comment:** the value of the every operator depends deeply on the time step. It can be used to do something not every step.
- **Examples:**

```
if every(2) {write "the time step is even";}
else {write "the time step is odd";}
```

[Top of the page](#)

exp

- Possible use:
 - OP(int) --- > float
 - OP(float) --- > 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](#),

[Top of the page](#)

—

fact

- Possible use:
 - OP(int) --- > int
- **Result:** Returns the factorial of the operand.
- **Special cases:**
 - if the operand is less than 0, fact returns 0.
- **Examples:**

```
int var0 <- fact(4);      // var0 equals 24
```

[Top of the page](#)

—

farthest_point_to

- Possible use:
 - geometry OP 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 closest
point of geom to pt
```

- **See also:** [any_location_in](#) , [any_point_in](#) , [closest_points_with](#) , [points_at](#) ,

[Top of the page](#)

—

file

- Possible use:
 - OP(string) --- > file

- string OP container --- > file
- **Result:** Creates a file in read/write mode, setting its contents to the container passed in parameteropens a file in read only mode, creates a GAML file object, and tries to determine and store the file content in the contents attribute.
- **Comment:** 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.pngThe file should have a supported extension, see file type deifnition for supported file extensions.
- **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](#) ,

[Top of the page](#)

file_exists

- Possible use:
 - OP(string) --- > bool
- **Result:** Test whether the parameter is the path to an existing file.

[Top of the page](#)

first

- Possible use:
 - OP(container) --- > [ValueType]
 - OP(string) --- > string
 - int OP container --- > list
- **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 list, first returns the first element of the list, or nil if the list is empty

```
int var0 <- first ([1, 2, 3]);      // var0 equals 1
```

- if it is a string, first returns a string composed of its first character

```
string var1 <- first ('abce');      // var1 equals 'a'
```

- **See also:** [last](#),

[Top of the page](#)

first_with

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

flip

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

float

- Possible use:
 - OP(any) ---> float
- **Result:** Casts the operand into the type float

[Top of the page](#)

floor

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

—

folder

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

—

frequency_of

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

fuzzy_kappa

- Possible use:
 - ??? OP list --- > float
 - ??? OP 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).` Reference:
 Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software,
 21fuzzy 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
- **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)  

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])
```

[Top of the page](#)

fuzzy_kappa_sim

- Possible use:
 - ??? OP list --- > float
 - ??? OP 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).` 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, 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,

- **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],[0,1,0,0,0,0,0,0],[0,0,1,0,0,0,0,0],  
[0,0,0,1,0,0,0,0],[0,0,0,0,1,0,0,0],[0,0,0,0,0,1,0,0],  
[0,0,0,0,0,1,0,0],[0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,1]], 2)  
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],[0,1,0,0,0,0,0,0],[0,0,1,0,0,0,0,0],  
[0,0,0,1,0,0,0,0],[0,0,0,0,1,0,0,0],[0,0,0,0,0,1,0,0],  
[0,0,0,0,0,1,0,0],[0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,1]], 2,  
[1.0,3.0,2.0,2.0,4.0])
```

[Top of the page](#)

gaml_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type gaml. Allowed extensions are limited to gaml

[Top of the page](#)

gamma

- Possible use:
 - OP(float) --- > float

[Top of the page](#)

gamma_index

- Possible use:
 - OP(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: $\text{gamma} = e / (3 * (v - 2))$ - 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](#) ,

[Top of the page](#)

gauss

- Possible use:
 - OP(point) --- > float
 - float OP 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](#) ,

[Top of the page](#)

generate_barabasi_albert

- Possible use:

- species OP species --- > graph
- **Result:** 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:
- **Special cases:**
 - "edges_specy": the species of edges
 - "vertices_specy": the species of vertices
 - "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(
    yourNodeSpecy,
    yourEdgeSpecy,
    3,
    5,
    true);
```

- **See also:** [generate_watts_strogatz](#),

[Top of the page](#)

generate_complete_graph

- Possible use:
 - species OP species --- > graph
 - species OP species --- > graph
- **Result:** returns a fully connected graph.returns a fully connected graph.
- **Comment:** Arguments should include following elements:Arguments should include following elements:
- **Special cases:**
 - "vertices_specy": the species of vertices
 - "edges_specy": 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?
 - "vertices_specy": the species of vertices
 - "edges_specy": the species of edges

- "size": the graph will contain size nodes.
- "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(
    myVertexSpecy,
    myEdgeSpecy,
    10,
    true);
```

- **See also:** [generate_barabasi_albert](#) , [generate_watts_strogatz](#) ,

[Top of the page](#)

generate_watts_strogatz

- Possible use:
 - species OP species --- > graph
- **Result:** 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:
- **Special cases:**
 - "vertices_specy": the species of vertices
 - "edges_specy": 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(
    myVertexSpecy,
    myEdgeSpecy,
    2,
    0.3,
    2,
```

```
true);
```

- **See also:** [generate_barabasi_albert](#),

[Top of the page](#)

geometric_mean

- Possible use:
 - OP(container) --- > float
- **Result:** the geometric mean of the elements of the operand. See < A href="http://en.wikipedia.org/wiki/Geometric_mean">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](#) ,

[Top of the page](#)

geometry

- Possible use:
 - OP(any) --- > geometry
- **Result:** Casts the operand into the type geometry

[Top of the page](#)

geometry_collection

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

get

- Possible use:
 - geometry OP string ---> unknown
 - agent OP string ---> unknown
- **Result:** Reads an attribute of the specified geometry (left operand). The attribute name is specified by the right operand. Reads an attribute of the specified agent (left operand). The attribute name is specified by the right operand.
- **Examples:**

```
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
string agent_name <- an_agent get('name');           // reads then 'name'
attribute of an_agent then assigns the returned value to the agent_name
variable
```

[Top of the page](#)

gml_from_wfs

- Possible use:
 - string OP string ---> msi.gama.util. [GamaList]< msi.gama.util. [GamaList]<
 java.lang.Object >>
- **Result:** WMS: A simple call to WFS/GML2

[Top of the page](#)

GPU_path_between

- Possible use:
 - graph OP geometry --- > path
- **Result:** The shortest path between a list of two objects in a graph computed with GPU
- **Examples:**

```
path var0 <- my_graph GPU_path_between (ag1:: ag2);      // var0 equals A
path between ag1 and ag2
```

[Top of the page](#)

graph

- Possible use:
 - OP(any) --- > graph
- **Result:** Casts the operand into the type graph

[Top of the page](#)

grayscale

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

grid_at

- Possible use:
 - species OP 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
```

[Top of the page](#)

grid_cells_to_graph

- Possible use:
 - OP(container) --- > graph
- **Result:** creates a graph from a list of cells (operand). An edge is created between neighbours.
- **Examples:**

```
my_cell_graph<-grid_cells_to_graph(cells_list)
```

[Top of the page](#)

grid_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type grid. Allowed extensions are limited to asc

[Top of the page](#)

group_by

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

harmonic_mean

- Possible use:
 - OP(container) --- > float
- **Result:** the harmonic mean of the elements of the operand. See < A href="http://en.wikipedia.org/wiki/Harmonic_mean">[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](#) ,

[Top of the page](#)

hemisphere

- Possible use:
 - float OP float --- > geometry
- **Result:** An hemisphere geometry which radius is equal to the operand.
- **Comment:** the centre of the hemisphere 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 <- hemisphere(10,0.5);      // var0 equals a geometry as a
circle of radius 10 but displays an hemisphere.
```

- **See also:** [around](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#) , [hemisphere](#) , [pie3D](#) ,

[Top of the page](#)

hexagon

- Possible use:
 - OP(float) --- > geometry
 - OP(point) --- > geometry
- **Result:** A hexagon geometry which the given width and height
- **Comment:** the centre of the hexagon is by default the location of the current agent in which has been called this operator.the centre 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);      // var0 equals a geometry as a hexagon of
width of 10 and height of 10.
geometry var1 <- hexagon({10,5});    // var1 equals a geometry as a
hexagon of width of 10 and height of 5.
```

- **See also:** [around](#) , [circle](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [triangle](#) ,

[Top of the page](#)

hierarchical_clustering

- Possible use:
 - container<agent> OP float --- > list
- **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:**

```
list 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](#),

[Top of the page](#)

hsb

- Possible use:
 - float OP float --- > rgb
 - float OP float --- > rgb
 - float OP 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](#),

[Top of the page](#)

hypot

- Possible use:
 - float OP float --- > float
- **Result:** Returns $\sqrt{x^2 + y^2}$ 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)
```

[Top of the page](#)

image_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type image. Allowed extensions are limited to tif, tiff, jpg, jpeg, png, gif, pict, bmp

[Top of the page](#)

image_from_direct_wms

- Possible use:
 - string OP string --- > file
- **Result:** WMS: A simple call to WMS

[Top of the page](#)

image_from_wms

- Possible use:
 - string OP string --- > file
- **Result:** WMS: A simple call to WMS

[Top of the page](#)

in

- Possible use:
 - string OP string --- > bool
 - unknown OP 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](#) ,

[Top of the page](#)

in_degree_of

- Possible use:
 - graph OP 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](#) ,

[Top of the page](#)

in_edges_of

- Possible use:
 - graph OP unknown --- > list
- **Result:** returns the list of the in-edges of a vertex (right-hand operand) in the graph given as left-hand operand.
- **Examples:**

```
list var1 <- graphFromMap in_edges_of node({12,45});           // var1 equals
[LineString]
```

- **See also:** [out_edges_of](#),

[Top of the page](#)

incomplete_beta

- Possible use:
 - float OP float --- > float

[Top of the page](#)

incomplete_gamma

- Possible use:
 - float OP float --- > float

[Top of the page](#)

incomplete_gamma_complement

- Possible use:
 - float OP float --- > float

[Top of the page](#)

index_by

- Possible use:
 - container OP 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]
```

[Top of the page](#)

index_of

- Possible use:
 - list OP unknown --- > int
 - species OP unknown --- > int
 - msigama.util. [GamaMap]< ?,? > OP unknown --- > unknown
 - matrix OP unknown --- > point
 - string OP string --- > int
- **Result:** the index of the first occurrence of the right operand in the left operand container the index of the first occurrence 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 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}
```

- if both operands are strings, returns the index within the left-hand string of the first occurrence of the given right-hand string

```
int var4 <- "abcabcabc" index_of "ca";      // var4 equals 2
```

- **Examples:**

```
unknown var0 <- [1::2, 3::4, 5::6] index_of 4;      // var0 equals 3
```

- **See also:** [at](#) , [last_index_of](#) ,

[Top of the page](#)

inside

- Possible use:
 - `msi.gama.util.IContainer < ?,? extends msi.gama.metamodel.shape.IShape > OP unknown ---> list<agent>`
- **Result:** A list of agents among the left-operand list, species or meta-population (addition of species), covered by the operand (casted as a geometry).
- **Examples:**

```
list<agent> 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<agent> 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:** [neighbours_at](#) , [neighbours_of](#) , [closest_to](#) , [overlapping](#) , [agents_overlapping](#) , [agents_inside](#) , [agent_closest_to](#) ,

[Top of the page](#)

int

- Possible use:
 - `OP(any) ---> int`
- **Result:** Casts the operand into the type int

[Top of the page](#)

inter

- Possible use:
 - container OP container --- > list
 - geometry OP geometry --- > geometry
- **Result:** the intersection of the two operandsA geometry resulting from the intersection between the two geometries
- **Comment:** both containers are transformed into sets (so without duplicated element, cf. remove_duplicates operator) before the set intersection is computed.
- **Special cases:**
 - if an operand is a graph, it will be transformed into the set of its nodes
 - returns nil if one of the operands is nil
 - if an operand is a map, it will be transformed into the set of its values

```
list var0 <- [1::2, 3::4, 5::6] inter [2,4];      // var0 equals [2,4]
list var1 <- [1::2, 3::4, 5::6] inter [1,3];      // var1 equals []
```

- if an operand is a matrix, it will be transformed into the set of the lines

```
list var2 <- matrix([[1,2,3],[4,5,4]]) inter [3,4];    // var2 equals
[3,4]
```

- **Examples:**

```
list var3 <- [1,2,3,4,5,6] inter [2,4];      // var3 equals [2,4]
list var4 <- [1,2,3,4,5,6] inter [0,8];      // var4 equals []
geometry var5 <- square(10) inter circle(5);   // var5 equals circle(5)
```

- **See also:** [remove_duplicates](#) , [union](#) , [+](#) , [-](#) ,

[Top of the page](#)

interleave

- Possible use:
 - OP(container) --- > list
- **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:**

```
list var0 <- interleave([1,2,4,3,5,7,6,8]);      // var0 equals
[1,2,4,3,5,7,6,8]
list var1 <- interleave([[['e11','e12','e13'],
[['e21','e22','e23'], ['e31','e32','e33']]));    // var1 equals
['e11','e21','e31','e12','e22','e32','e13','e23','e33']
```

[Top of the page](#)

internal_at

- Possible use:
 - geometry OP list ---> unknown
 - msi.gama.util.IContainer <[KeyType] , [ValueType]> .Addressable <[KeyType] , [ValueType]> OP msi.gama.util.IList <[KeyType]> ---> [ValueType]
- **Result:** For internal use only. Corresponds to the implementation of the access to containers with [index] For internal use only. Corresponds to the implementation of the access to containers with [index]
- **See also:** [at](#) ,

[Top of the page](#)

internal_zero_order_equation

- Possible use:
 - OP(any expression) ---> float

[Top of the page](#)

intersection

Same signification as [inter](#) operator.

[Top of the page](#)

intersects

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

—

is

- Possible use:
 - unknown OP 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
```

[Top of the page](#)

—

is_CSV

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a csv file.

[Top of the page](#)

is_finite

- Possible use:
 - OP(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
```

[Top of the page](#)

is_gaml

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a gaml file.

[Top of the page](#)

is_grid

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a grid file.

[Top of the page](#)

is_image

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a image file.

[Top of the page](#)

is_number

- Possible use:
 - OP(string) --- > bool
 - OP(float) --- > bool
- **Result:** tests whether the operand represents a numerical value Returns whether the argument is a real number or not
- **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("test");      // var0 equals false
bool var1 <- is_number("123.56");    // var1 equals true
bool var2 <- is_number("-1.2e5");    // var2 equals true
bool var3 <- is_number("1,2");       // var3 equals false
bool var4 <- is_number("#12FA");    // var4 equals true
bool var5 <- is_number(4.66);       // var5 equals true
bool var6 <- is_number(#infinity);  // var6 equals true
bool var7 <- is_number(#nan);       // var7 equals false
```

[Top of the page](#)

is_obj

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a obj file.

[Top of the page](#)

is_obj

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a obj file.

[Top of the page](#)

is_osm

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a osm file.

[Top of the page](#)

is_pgm

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a pgm file.

[Top of the page](#)

is_property

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a property file.

[Top of the page](#)

is_R

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a R file.

[Top of the page](#)

is_shape

- Possible use:
 - OP(any) --- > bool

- **Result:** Tests whether the operand is a shape file.

[Top of the page](#)

is_skill

- Possible use:
 - unknown OP 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
```

[Top of the page](#)

is_svg

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a svg file.

[Top of the page](#)

is_text

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a text file.

[Top of the page](#)

is_threeds

- Possible use:

- OP(any) --- > bool
- **Result:** Tests whether the operand is a threeds file.

[Top of the page](#)

is_threeds

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a threeds file.

[Top of the page](#)

is_URL

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a URL file.

[Top of the page](#)

is_xml

- Possible use:
 - OP(any) --- > bool
- **Result:** Tests whether the operand is a xml file.

[Top of the page](#)

kappa

- Possible use:
 - list OP list --- > float
 - list OP list --- > float
- **Result:** 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.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.

- **Examples:**

```
kappa([cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
[cat1,cat2,cat3])  
float var1 <- kappa([1,3,5,1,5],[1,1,1,1,5],[1,3,5]); // var1 equals  
the similarity between 0 and 1  
float var2 <- kappa([1,1,1,1,5],[1,1,1,1,5],[1,3,5]); // var2 equals  
1.0  
kappa([cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
[cat1,cat2,cat3], [1.0, 2.0, 3.0, 1.0, 5.0])
```

[Top of the page](#)

—

kappa_sim

- Possible use:
 - list OP list --- > float
 - list OP list --- > float
- **Result:** 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)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).

- **Examples:**

```
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])  
kappa([cat1,cat1,cat2,cat2,cat2],[cat2,cat1,cat2,cat1,cat3],  
[cat2,cat1,cat2,cat3,cat3], [cat1,cat2,cat3])
```

[Top of the page](#)

—

kurtosis_1

- Possible use:
 - OP(container) --- > float

[Top of the page](#)

kurtosis_2

- Possible use:
 - float OP float --- > float

[Top of the page](#)

R_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type R. Allowed extensions are limited to r

[Top of the page](#)

TGauss

Same signification as [truncated_gauss](#) operator.

[Top of the page](#)

URL_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type URL. Allowed extensions are limited to txt

[Top of the page](#)

6.6.6 Operators (L to Z)

Operators (L to Z)

 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 infix notation (i.e. `operand1 operator_symbol operand1`). The ternary functional if-else operator, `? :`, uses a special infix 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 infix 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, ..., operandn)'`) or a special infix syntax with the first operand on the left-hand side of the operator name (e.g. `'operand1 operator_name(operand2, ..., operandn)'`) 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 species1 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). [Top of the page](#)

—

Table of Contents

—

Operators by categories

—

3D

- [box](#) , [cone3D](#) , [cube](#) , [cylinder](#) , [dem](#) , [hexagon](#) , [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](#) ,

—

Casting operators

- [as](#) , [as_int](#) , [as_matrix](#) , [is](#) , [is_skill](#) , [list_with](#) , [matrix_with](#) , [species](#) , [to_gaml](#) , [topology](#) ,

—

Color-related operators

- [-](#) , [/](#) , [^](#) , [+](#) , [blend](#) , [grayscale](#) , [hsb](#) , [rgb](#) , [rnd_color](#) ,

Comparison operators

- `!=`, `<`, `<=`, `=`, `>`, `>=`, `between`,

Containers-related operators

- `-`, `::`, `+`, `accumulate`, `among`, `at`, `collect`, `contains`, `contains_all`, `contains_any`, `count`, `empty`, `first`, `first_with`, `group_by`, `in`, `index_by`, `inter`, `interleave`, `internal_at`, `last`, `last_with`, `length`, `max`, `max_of`, `min`, `min_of`, `mul`, `one_of`, `remove_duplicates`, `reverse`, `shuffle`, `sort_by`, `sum`, `union`, `where`, `with_max_of`, `with_min_of`,

DescriptiveStatistics

- `auto_correlation`, `correlation`, `covariance`, `durbin_watson`, `kurtosis_1`, `kurtosis_2`, `moment`, `quantile`, `quantile_inverse`, `rank_interpolated`, `rms`, `skew_1`, `skew_2`, `variance1`, `variance2`,

Distributions

- `beta`, `binomial_coeff`, `binomial_complemented`, `binomial_sum`, `chi_square`, `chi_square_complemented`, `gamma`, `incomplete_beta`, `incomplete_gamma`, `incomplete_gamma_complement`, `log_gamma`, `normal_area`, `normal_density`, `normal_inverse`, `pValue_for_fStat`, `pValue_for_tStat`, `student_area`, `student_t_inverse`,

Driving operators

- `as_driving_graph`,

EDP-related operators

- `diff` , `diff2` , `internal_zero_order_equation` ,
-

Files-related operators

- `crs` , `csv_file` , `file` , `file_exists` , `folder` , `gaml_file` , `get` , `grid_file` , `image_file` , `is_csv` , `is_gaml` , `is_grid` , `is_image` , `is_obj` , `is_obj` , `is_osm` , `is_pgm` , `is_property` , `is_R` , `is_shape` , `is_svg` , `is_text` , `is_threeds` , `is_threeds` , `is_URL` , `is_xml` , `new_folder` , `obj_file` , `obj_file` , `osm_file` , `pgm_file` , `property_file` , `R_file` , `read` , `shape_file` , `svg_file` , `text_file` , `threeds_file` , `threeds_file` , `URL_file` , `writable` , `xml_file` ,
-

FIPA-related operators

- `conversation` , `message` ,
-

Graphs-related operators

- `add_edge` , `add_node` , `agent_from_geometry` , `all_pairs_shortest_path` , `alpha_index` , `as_distance_graph` , `as_edge_graph` , `as_intersection_graph` , `as_path` , `beta_index` , `betweenness_centrality` , `connected_components_of` , `connectivity_index` , `contains_edge` , `contains_vertex` , `CPU_path_between` , `degree_of` , `directed` , `edge` , `edge_between` , `edges` , `gamma_index` , `generate_barabasi_albert` , `generate_complete_graph` , `generate_watts_strogatz` , `GPU_path_between` , `grid_cells_to_graph` , `in_degree_of` , `in_edges_of` , `layout` , `load_graph_from_file` , `load_shortest_paths` , `nb_cycles` , `neighbours_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` ,
-

Iterator operators

- `accumulate`, `as_map`, `collect`, `count`, `first_with`, `frequency_of`, `group_by`, `index_by`,
`last_with`, `max_of`, `min_of`, `sort_by`, `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`, `new_predicate`,
-

Matrix-related operators

- `-`, `/`, `.`, `*`, `+`, `append_horizontally`, `append_vertically`, `column_at`, `columns_list`, `index_of`,
`last_index_of`, `row_at`, `rows_list`, `shuffle`,
-

OpenGIS

- `gml_from_wfs`, `image_from_direct_wms`, `image_from_wms`, `read_json_rest`,

Path-related operators

- `agent_from_geometry`, `all_pairs_shortest_path`, `as_path`, `CPU_path_between`,
`GPU_path_between`, `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`, `shuffle`, `truncated_gauss`,

Shape

- `antislice`, `box`, `circle`, `cone`, `cone3D`, `cube`, `cylinder`, `envelope`, `geometry_collection`,
`hemisphere`, `hexagon`, `line`, `link`, `pacman`, `plan`, `polygon`, `polyhedron`, `pyramid`,
`rectangle`, `rgbcube`, `rgbtriangle`, `slice`, `sphere`, `spherical_pie`, `square`, `teapot`, `triangle`,

Spatial operators

- `-`, `*`, `+`, `add_point`, `agent_closest_to`, `agents_at_distance`, `agents_inside`,
`agents_overlapping`, `angle_between`, `antislice`, `any_location_in`, `around`, `as_4_grid`,
`as_grid`, `as_hexagonal_grid`, `at_distance`, `at_location`, `box`, `circle`, `clean`,
`closest_points_with`, `closest_to`, `cone`, `cone3D`, `convex_hull`, `covers`, `crosses`,
`crs`, `cube`, `cylinder`, `dem`, `direction_between`, `disjoint_from`, `distance_between`,
`distance_to`, `envelope`, `farthest_point_to`, `geometry_collection`, `hemisphere`,
`hexagon`, `hierarchical_clustering`, `inside`, `inter`, `intersects`, `line`, `link`, `masked_by`,
`neighbours_at`, `neighbours_of`, `overlapping`, `overlaps`, `pacman`, `partially_overlaps`,
`path_between`, `path_to`, `plan`, `points_at`, `points_on`, `polygon`, `polyhedron`, `pyramid`,

rectangle , rgb_to_xyz , rgbcube , rgbtriangle , rotated_by , round , scaled_to , set_z , simple_clustering_by_distance , simplification , skeletonize , slice , sphere , spherical_pie , split_at , split_geometry , split_lines , square , teapot , to_rectangles , to_squares , touches , towards , transformed_by , translated_by , triangle , triangulate , union , voronoi , with_precision , without_holes ,

Spatial properties operators

- covers , crosses , intersects , partially_overlaps , touches ,

Spatial queries operators

- agent_closest_to , agents_at_distance , agents_inside , agents_overlapping , at_distance , closest_to , inside , neighbours_at , neighbours_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 , rotated_by , scaled_to , simplification , skeletonize , split_geometry , split_lines , 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

- `clustering_cobweb`, `clustering_DBScan`, `clustering_em`, `clustering_farthestFirst`,
`clustering_simple_kmeans`, `clustering_xmeans`, `corR`, `frequency_of`, `geometric_mean`,
`harmonic_mean`, `hierarchical_clustering`, `max`, `mean`, `mean_deviation`, `meanR`, `median`,
`min`, `mul`, `simple_clustering_by_distance`, `standard_deviation`, `sum`, `variance`,

—

Strings-related operators

- `+`, `<`, `<=`, `>`, `>=`, `as_date`, `as_time`, `at`, `char`, `contains`, `contains_all`, `contains_any`,
`copy_between`, `empty`, `first`, `in`, `index_of`, `is_number`, `last`, `last_index_of`, `length`,
`replace`, `reverse`, `sample`, `shuffle`, `split_with`,

—

System

- `.`, `copy`, `dead`, `eval_gaml`, `every`, `user_input`,

—

Time-related operators

- `as_date`, `as_time`,

—

Types-related operators

- `agent`, `bool`, `container`, `float`, `geometry`, `graph`, `int`, `list`, `map`, `matrix`, `pair`, `path`,
`string`, `unknown`,

—

User control operators

- `user_input`,
-

Water level operators

- `water_area_for`, `water_level_for`, `water_polylines_for`,
-

Operators

last

- Possible use:
 - `OP(string) ---> string`
 - `OP(container) ---> [ValueType]`
 - `int OP container ---> list`
- **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](#),

[Top of the page](#)

—

last_index_of

- Possible use:
 - matrix OP unknown --- > point
 - species OP unknown --- > int
 - list OP unknown --- > int
 - string OP string --- > int
 - msi.gama.util. [GamaMap]< ?,? > OP unknown --- > unknown
- **Result:** the index of the last occurrence 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 the left operand is a matrix, last_index_of returns the index as a point

```
point var0 <- matrix([[1,2,3],[4,5,4]]) last_index_of 4;           // var0 equals {1.0,2.0}
```

- if the left operand is a list, last_index_of returns the index as an integer

```
int var1 <- [1,2,3,4,5,6] last_index_of 4;           // var1 equals 3
int var2 <- [4,2,3,4,5,4] last_index_of 4;           // var2 equals 5
```

- if both operands are strings, returns the index within the left-hand string of the rightmost occurrence of the given right-hand string

```
int var3 <- "abcababc" last_index_of "ca";           // 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](#) , [index_of](#) , [last_index_of](#) ,

[Top of the page](#)

last_with

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

layout

- Possible use:
 - graph OP string --- > graph
 - graph OP string --- > graph
 - graph OP string --- > graph
- **Result:** layouts a GAMA graph.

[Top of the page](#)

length

- Possible use:
 - OP(container) --- > int
 - OP(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
```

[Top of the page](#)

line

- Possible use:
 - OP(container<geometry>) --- > geometry
 - container<geometry> OP float --- > geometry
- **Result:** A polyline geometry from the given list of points.A polyline geometry from the given list of points represented as a cylinder of radius r.
- **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 var1 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}],0.2);      //  
var1 equals a polyline geometry composed of the 4 points.
```

- **Examples:**

```
geometry var0 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}]);      // var0  
equals a polyline geometry composed of the 4 points.
```

- **See also:** [around](#) , [circle](#) , [cone](#) , [link](#) , [norm](#) , [point](#) , [polygone](#) , [rectangle](#) , [square](#) , [triangle](#) ,

[Top of the page](#)

link

- Possible use:
 - OP(pair) ---> geometry
- **Result:** A link between the 2 elements of the pair.
- **Comment:** The geometry of the link is the intersection of the two geometries when they intersect, and a line between their centroids when they do not.
- **Special cases:**
 - if the operand is nil, link returns a point {0,0}
 - if one of the elements of the pair is a list of geometries or a species, link will consider the union of the geometries or of the geometry of each agent of the species
- **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](#) ,

[Top of the page](#)

list

- Possible use:
 - OP(any) ---> list
- **Result:** Casts the operand into the type list

[Top of the page](#)

list_with

- Possible use:
 - int OP any expression --- > list
- **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](#) ,

[Top of the page](#)

ln

- Possible use:
 - OP(float) --- > float
 - OP(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](#) ,

[Top of the page](#)

load_graph_from_file

- Possible use:
 - OP(string) --- > graph
 - string OP string --- > graph
 - string OP file --- > graph
 - string OP species --- > graph
 - string OP file --- > graph
 - string OP string --- > graph
 - string OP string --- > 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 GAMAloads 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_specy": 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_specy": 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

```
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

```
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_specy": 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");
```

- "filename": the filename of the file containing the network, "edges_specy": 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 );
```

- **Examples:**

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
    "pajek",
    "./example_of_Pajek_file",
    myVertexSpecy,
    myEdgeSpecy);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
    "pajek",
    "./example_of_Pajek_file",
    myVertexSpecy,
    myEdgeSpecy , true);
```

[Top of the page](#)

load_shortest_paths

- Possible use:
 - graph OP 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
```

[Top of the page](#)

log

- Possible use:
 - OP(float) --- > float
 - OP(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](#) ,

[Top of the page](#)

log_gamma

- Possible use:
 - OP(float) --- > float

[Top of the page](#)

map

- Possible use:
 - OP(any) --- > map
- **Result:** Casts the operand into the type map

[Top of the page](#)

masked_by

- Possible use:
 - geometry OP container<agent> --- > geometry
 - geometry OP container<agent> --- > 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.
```

[Top of the page](#)

matrix

- Possible use:
 - OP(any) --- > matrix
- **Result:** Casts the operand into the type matrix

[Top of the page](#)

matrix_with

- Possible use:
 - point OP unknown --- > 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](#) ,

[Top of the page](#)

max

- Possible use:
 - OP(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 or 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](#),

[Top of the page](#)

max_of

- Possible use:
 - container OP 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 9
```

- **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](#),

[Top of the page](#)

mean

- Possible use:
 - OP(any expression) --- > 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
- **Examples:**

```
unknown var0 <- mean ([4.5, 3.5, 5.5, 7.0]);           // var0 equals 5.125
```

- **See also:** [sum](#) ,

[Top of the page](#)

mean_deviation

- Possible use:
 - OP(container) --- > float
- **Result:** the deviation from the mean of all the elements of the operand. See < A href= "http://en.wikipedia.org/wiki/Absolute_deviation"> 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](#) ,

[Top of the page](#)

meanR

- Possible use:
 - OP(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
```

[Top of the page](#)

median

- Possible use:
 - OP(container) --- > float
- **Result:** the median of all the elements of the operand.
- **Comment:** The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.
- **Examples:**

```
float var0 <- median ([4.5, 3.5, 5.5, 7.0]);           // var0 equals 5.0
```

- **See also:** [mean](#),

[Top of the page](#)

message

- Possible use:
 - OP(unknown) --- > msi.gaml.extensions.fipa.Message
- **Result:** to be added

[Top of the page](#)

min

- Possible use:
 - OP(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](#),

[Top of the page](#)

min_of

- Possible use:
 - container OP 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](#),

[Top of the page](#)

—

mod

- Possible use:
 - int OP 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](#),

[Top of the page](#)

—

moment

- Possible use:
 - container OP int --- > float

[Top of the page](#)

—

mul

- Possible use:
 - OP(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](#),

[Top of the page](#)

nb_cycles

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

neighbours_at

- Possible use:
 - geometry OP float ---> list

- **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 neighbourhood is the one of the left-operand if this one is an agent; otherwise the one of the agent applying the operator.
- **Examples:**

```
list var0 <- (self neighbours_at (10));      // var0 equals all the agents located at a distance lower or equal to 10 to the agent applying the operator.
```

- **See also:** [neighbours_of](#) , [closest_to](#) , [overlapping](#) , [agents_overlapping](#) , [agents_inside](#) , [agent_closest_to](#) , [at_distance](#) ,

[Top of the page](#)

neighbours_of

- Possible use:
 - topology OP agent ---> list
 - graph OP unknown ---> list
 - topology OP geometry ---> list
- **Result:** a list, containing all the agents 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 located at a distance inferior or equal to the third argument to the second argument (agent, geometry or point) considering the first operand topology.

```
list var0 <- neighbours_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:**

```
list var1 <- topology(self) neighbours_of self;      // var1 equals returns all the agents located at a distance lower or equal to 1 to the agent applying the operator considering its topology.  
list var2 <- graphEpidemio neighbours_of (node(3));      // var2 equals [node0,node2]  
list var3 <- graphFromMap neighbours_of node({12,45});      // var3 equals [{1.0,5.0},{34.0,56.0}]
```

- **See also:** [neighbours_at](#) , [closest_to](#) , [overlapping](#) , [agents_overlapping](#) , [agents_inside](#) , [agent_closest_to](#) , [predecessors_of](#) , [successors_of](#) ,

[Top of the page](#)

new_folder

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

new_predicate

- Possible use:
 - string OP unknown --- > map<string,unknown>
- **Result:** returns the predicate with the given name, value and parameters
- **Examples:**

```
GamaList mypredicate<-new_predicate(PredName,PredValue,[par1::val1])
GamaList OnAB<-new_predicate("On",true,[ "sub":::blocka,"on":::blockb])
```

- **See also:** [simple_bdi](#) ,

[Top of the page](#)

node

- Possible use:

- OP(unknown) --- > unknown
- unknown OP float --- > unknown

[Top of the page](#)

—

nodes

- Possible use:
 - OP(container) --- > container

[Top of the page](#)

—

norm

- Possible use:
 - OP(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
```

[Top of the page](#)

—

normal_area

- Possible use:
 - float OP float --- > float

[Top of the page](#)

—

normal_density

- Possible use:
 - float OP float --- > float

[Top of the page](#)

normal_inverse

- Possible use:
 - float OP float --- > float

[Top of the page](#)

not

Same signification as `!` operator.

[Top of the page](#)

obj_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type obj. Allowed extensions are limited to obj

[Top of the page](#)

obj_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type obj. Allowed extensions are limited to obj

[Top of the page](#)

of

Same signification as `.` operator.

[Top of the page](#)

of_generic_species

- Possible use:
 - container OP species --- > list
- **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 {}
list var2 <- [sous_test(0),sous_test(1),test(2),test(3)] of_generic_species
test;      // var2 equals [sous_test0,sous_test1,test2,test3]
list var3 <- [sous_test(0),sous_test(1),test(2),test(3)] of_generic_species
sous_test;      // var3 equals [sous_test0,sous_test1]
list var4 <- [sous_test(0),sous_test(1),test(2),test(3)] of_species test;
    // var4 equals [test2,test3]
list 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](#),

[Top of the page](#)

of_species

- Possible use:
 - container OP species --- > list
- **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:**

```
list var0 <- (self neighbours_at 10) of_species (species (self));      //
var0 equals all the neighbouring agents of the same species.
```

```
list var1 <- [test(0),test(1),node(1),node(2)] of_species test;      // var1
equals [test0,test1]
```

- **See also:** [of_generic_species](#),

[Top of the page](#)

one_of

- Possible use:
 - OP(container) ---> [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](#),

[Top of the page](#)

Or

- Possible use:
 - bool OP 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](#) ,

[Top of the page](#)

osm_file

- Possible use:
 - string OP msi.gama.util. [GamaMap]< java.lang.String,msi.gama.util. [GamaList]> --- > file
 - string OP msi.gama.util. [GamaMap]< java.lang.String,msi.gama.util. [GamaList]> --- > file
- **Result:** opens a file that is a kind of OSM file with some filtering.opens a file that 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](#) , [properties](#) , [image](#) , [text](#) ,

[Top of the page](#)

out_degree_of

- Possible use:
 - graph OP 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](#),

[Top of the page](#)

out_edges_of

- Possible use:
 - graph OP unknown --- > list
- **Result:** returns the list of the out-edges of a vertex (right-hand operand) in the graph given as left-hand operand.
- **Examples:**

```
list var1 <- graphFromMap out_edges_of (node(3)); // var1 equals 3
```

- **See also:** [in_edges_of](#),

[Top of the page](#)

overlapping

- Possible use:
 - `msi.gama.util.IContainer < ?,? extends msi.gama.metamodel.shape.IShape > OP unknown --- > list<agent>`
- **Result:** A list of agents among the left-operand list, species or meta-population (addition of species), overlapping the operand (casted as a geometry).
- **Examples:**

```
list<agent> 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:** [neighbours_at](#) , [neighbours_of](#) , [agent_closest_to](#) , [agents_inside](#) , [closest_to](#) , [inside](#) , [agents_overlapping](#) ,

[Top of the page](#)

overlaps

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

pacman

- Possible use:
 - OP(float) --- > geometry
 - float OP float --- > geometry
- **Result:** An pacman geometry which radius is equal to first argument. An pacman geometry with a dynamic opening mouth which radius is equal to first argument.
- **Comment:** the centre of the sphere is by default the location of the current agent in which has been called this operator. 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.
 - returns a point if the operand is lower or equal to 0.
- **Examples:**

```
geometry var0 <- pacman(1);      // var0 equals a geometry as a circle of
radius 10 but displays a sphere.
geometry var1 <- pacman(1,0.2);    // var1 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](#) , [hemisphere](#) , [pie3D](#) ,

[Top of the page](#)

pair

- Possible use:
 - OP(any) --- > pair
- **Result:** Casts the operand into the type pair

[Top of the page](#)

partially_overlaps

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

path

- Possible use:
 - OP(any) --- > path
- **Result:** Casts the operand into the type path

[Top of the page](#)

path_between

- Possible use:
 - topology OP container<geometry> --- > path
 - graph OP geometry --- > path
- **Result:** The shortest path between a list of two objects in a graph
- **Examples:**

```
path var0 <- path_between (my_graph, ag1, ag2);      // var0 equals A path
between ag1 and ag2
path var1 <- my_topology path_between [ag1, ag2];    // var1 equals A path
between ag1 and ag2
```

- **See also:** [towards](#) , [direction_to](#) , [distance_between](#) , [direction_between](#) , [path_to](#) , [distance_to](#)

[Top of the page](#)

path_to

- Possible use:
 - point OP point ---> path
 - geometry OP geometry ---> 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](#) ,

[Top of the page](#)

paths_between

- Possible use:
 - graph OP pair ---> 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
```

[Top of the page](#)

percent_absolute_deviation

- Possible use:
 - list<float> OP 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])
```

[Top of the page](#)

percentile

Same signification as [quantile_inverse](#) operator.

[Top of the page](#)

pgm_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type pgm. Allowed extensions are limited to pgm

[Top of the page](#)

plan

- Possible use:
 - container<geometry> OP 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](#) ,

[Top of the page](#)

point

- Possible use:
 - float OP int --- > point
 - float OP float --- > point
 - int OP int --- > point
 - int OP float --- > point
 - int OP float --- > point
 - float OP float --- > point
 - float OP int --- > point
 - int OP int --- > point
 - int OP int --- > point
 - float OP int --- > point
 - float OP float --- > point
- **Result:** internal use only. Use the standard construction {x,y} instead.internal use only. Use the standard construction {x,y} instead.internal use only. Use the standard construction {x,y} instead.internal use only. Use the standard construction {x,y} instead.

[Top of the page](#)

points_at

- Possible use:
 - int OP 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](#) ,

[Top of the page](#)

points_on

- Possible use:
 - geometry OP float ---> list
- **Result:** A list of points of the operand-geometry distant from each other to the float right-operand .
- **Examples:**

```
list 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](#) ,

[Top of the page](#)

poisson

- Possible use:
 - OP(float) ---> int
- **Result:** A value from a random variable following a Poisson distribution (with the positive expected number of occurrence 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](#) ,

[Top of the page](#)

polygon

- Possible use:
 - OP(msi.gama.util.IContainer < ?,? extends msi.gama.metamodel.shape.IShape >) ---> 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](#) ,

[Top of the page](#)

polyhedron

- Possible use:

- container<geometry> OP 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](#) ,

[Top of the page](#)

polyline

Same signification as [line](#) operator.

[Top of the page](#)

polyplan

Same signification as [plan](#) operator.

[Top of the page](#)

—

predecessors_of

- Possible use:
 - graph OP unknown --- > list
- **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:**

```
list var1 <- graphEpidemio predecessors_of ({1,5});      // var1 equals []
list var2 <- graphEpidemio predecessors_of node({34,56}); // var2
equals [{12;45}]
```

- **See also:** [neighbours_of](#) , [successors_of](#) ,

[Top of the page](#)

—

product

Same signification as [mul](#) operator.

[Top of the page](#)

—

property_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type property. Allowed extensions are limited to properties

[Top of the page](#)

—

pValue_for_fStat

- Possible use:
 - float OP int --- > float

[Top of the page](#)

pValue_for_tStat

- Possible use:
 - float OP int --- > float

[Top of the page](#)

pyramid

- Possible use:
 - OP(float) --- > geometry
- **Result:** A square geometry which side size is given by the operand.
- **Comment:** the centre 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](#)

[Top of the page](#)

quantile

- Possible use:
 - container OP float --- > float

[Top of the page](#)

quantile_inverse

- Possible use:
 - container OP float --- > float

[Top of the page](#)

rank_interpolated

- Possible use:
 - container OP float --- > float

[Top of the page](#)

read

- Possible use:
 - OP(string) --- > unknown
 - OP(int) --- > 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.
unknown second_variable <- read (2);      // second_variable equals reads
the second variable of agent then assigns the returned value to the
'second_variable' variable.
```

[Top of the page](#)

read_json_rest

- Possible use:
 - string OP string --- > msi.gama.util. [GamaList]< msi.gama.util. [GamaList]< java.lang.Object >>
- **Result:** REST: Read data from RESTService

[Top of the page](#)

rectangle

- Possible use:
 - OP(point) --- > geometry
 - float OP float --- > geometry
- **Result:** A rectangle geometry which side sizes are given by the operands.
- **Comment:** the centre of the rectangle is by default the location of the current agent in which has been called this operator.the centre 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.
- **Examples:**

```
geometry var0 <- rectangle({10, 5});      // var0 equals a geometry as a
rectangle with width = 10 and heigh = 5.
geometry var1 <- rectangle(10, 5);      // var1 equals a geometry as a
rectangle with width = 10 and heigh = 5.
```

- **See also:** [around](#) , [circle](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [square](#) , [triangle](#) ,

[Top of the page](#)

reduced_by

Same signification as [-](#) operator.

[Top of the page](#)

remove_duplicates

- Possible use:
 - OP(container) --- > list
- **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

```
list var1 <- remove_duplicates([1::3,2::4,3::3,5::7]);      // var1 equals
[3,4,7]
```

- **Examples:**

```
list var0 <- remove_duplicates([3,2,5,1,2,3,5,5,5]);      // var0 equals
[3,2,5,1]
```

[Top of the page](#)

remove_node_from

- Possible use:
 - geometry OP 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)
```

[Top of the page](#)

replace

- Possible use:
 - string OP string --- > string
- **Result:** Returns the String resulting by replacing for the first operand all the sub-strings corresponding to 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'
```

[Top of the page](#)

reverse

- Possible use:
 - OP(string) ---> string
 - OP(container) ---> container
- **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 string, reverse returns a new string with characters in the reversed order

```
string var0 <- reverse ('abcd');      // var0 equals 'dcba'
```

- if it is a list, reverse returns a copy of the operand list with elements in the reversed order

```
container var1 <- reverse ([10,12,14]);      // var1 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 var2 <- reverse ([ 'k1'::44, 'k2'::32, 'k3'::12]);      // var2
equals [12::'k3', 32::'k2', 44::'k1']
```

- if it is a matrix, reverse returns a new matrix containing the transpose of the operand.

```
container var3 <- reverse(matrix([["c11","c12","c13"],
["c21","c22","c23"]]));      // var3 equals matrix([["c11","c21"],
["c12","c22"],["c13","c23"]])
```

[Top of the page](#)

rewire_n

- Possible use:
 - graph OP int --- > graph
- **Result:** rewrites 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 egdes rewired
```

- **See also:** [rewire_p](#),

[Top of the page](#)

rgb

- Possible use:
 - rgb OP float --- > rgb
 - rgb OP int --- > rgb
 - string OP int --- > rgb
 - int OP int --- > rgb
 - int OP int --- > rgb
 - int OP int --- > rgb
- **Result:** Returns a color defined by red, green, blue components and an alpha blending value.
- **Special cases:**
 - It can be used with a color and an alpha between 0 and 1
 - It can be used with r=red, g=greeb, b=blue (each between 0 and 255), a=alpha (between 0.0 and 1.0)
 - It can be used with r=red, g=greeb, b=blue, each between 0 and 255
 - It can be used with r=red, g=greeb, 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 a name of color and alpha (between 0 and 255)
- **Examples:**

```
rgb var0 <- rgb(rgb(255,0,0),0.5);      // var0 equals a light red color
rgb var1 <- rgb (255,0,0,0.5);      // var1 equals a light red color
rgb var2 <- rgb (255,0,0);      // var2 equals #red
rgb var3 <- rgb (255,0,0,125);      // var3 equals a light red color
```

```
rgb var5 <- rgb(rgb(255,0,0),125);      // var5 equals a light red color
rgb var6 <- rgb ("red");      // var6 equals rgb(255,0,0)
```

- **See also:** [hsb](#),

[Top of the page](#)

rgb_to_xyz

- Possible use:
 - OP(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
```

[Top of the page](#)

rgbcube

- Possible use:
 - OP(float) --- > geometry
- **Result:** A cube geometry which side size is equal to the operand.
- **Comment:** the centre 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](#)

[Top of the page](#)

rgbtriangle

- Possible use:
 - OP(float) --- > geometry
- **Result:** A triangle geometry which side size is given by the operand.
- **Comment:** the centre 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](#)

[Top of the page](#)

rms

- Possible use:
 - int OP float --- > float

[Top of the page](#)

rnd

- Possible use:
 - OP(float) --- > float
 - OP(point) --- > point
 - OP(int) --- > int
 - point OP point --- > point
 - int OP int --- > int
 - float OP float --- > float
 - float OP float --- > float
 - point OP point --- > point
 - int OP 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 float, returns an uniformly distributed float random number in [0.0, to]
- if the operand is a point, returns a point with three random float ordinates, each in the interval [0, ordinate of argument]

- **Examples:**

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

- **See also:** [flip](#),

[Top of the page](#)

rnd_choice

- Possible use:

- OP(list) ---> 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](#),

[Top of the page](#)

rnd_color

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

rotated_by

- Possible use:
 - geometry OP float --- > geometry
 - geometry OP int --- > geometry
- **Result:** 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 <- self rotated_by 45;      // var0 equals the geometry  
resulting from a 45 degrees rotation to the geometry of the agent applying  
the operator.
```

- **See also:** [transformed_by](#) , [translated_by](#) ,

[Top of the page](#)

round

- Possible use:
 - OP(float) --- > int
 - OP(point) --- > point
 - OP(int) --- > int

- **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](#) ,

[Top of the page](#)

row_at

- Possible use:
 - matrix OP 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](#) ,

[Top of the page](#)

rows_list

- Possible use:
 - OP(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](#),

[Top of the page](#)

—

sample

- Possible use:
 - OP(any expression) ---> string
 - string OP any expression ---> string

[Top of the page](#)

—

scaled_by

Same signification as `*` operator.

[Top of the page](#)

—

scaled_to

- Possible use:
 - geometry OP 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
```

[Top of the page](#)

—

select

Same signification as `where` operator.

[Top of the page](#)

set_z

- Possible use:
 - geometry OP msi.gama.util.IContainer < ?,java.lang.Double > --- > geometry
 - geometry OP int --- > 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];
```

- **See also:** [add_z](#),

[Top of the page](#)

shape_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type shape. Allowed extensions are limited to shp

[Top of the page](#)

shuffle

- Possible use:
 - OP(container) --- > list
 - OP(string) --- > string
 - OP(matrix) --- > matrix
- **Result:** The elements of the operand in random order.
- **Special cases:**
 - if the operand is empty, returns an empty list (or string, matrix)
- **Examples:**

```
list var0 <- shuffle ([12, 13, 14]);      // var0 equals [14,12,13] (for example)
string var1 <- shuffle ('abc');        // var1 equals 'bac' (for example)
matrix var2 <- shuffle (matrix([["c11","c12","c13"], ["c21","c22","c23"]]));
// var2 equals matrix([["c12","c21","c11"], ["c13","c22","c23"]]) (for example)
```

- **See also:** [reverse](#),

[Top of the page](#)

signum

- Possible use:
 - OP(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
```

[Top of the page](#)

simple_clustering_by_distance

- Possible use:
 - container<agent> OP 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](#),

[Top of the page](#)

simple_clustering_by_envelope_distance

Same signification as [simple_clustering_by_distance](#) operator.

[Top of the page](#)

simplification

- Possible use:
 - geometry OP 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-Peucker algorithm on the
geometry of the agent applying the operator with a tolerance distance of
0.1.
```

[Top of the page](#)

sin

- Possible use:
 - OP(float) ---> float
 - OP(int) ---> 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
float var1 <- sin (0);      // var1 equals 0.0
```

- **See also:** [cos](#) , [tan](#) ,

[Top of the page](#)

sin_rad

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

skeletonize

- Possible use:
 - OP(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.
```

[Top of the page](#)

skew_1

- Possible use:
 - OP(container) ---> float

[Top of the page](#)

skew_2

- Possible use:
 - float OP float --- > float

[Top of the page](#)

slice

- Possible use:
 - float OP float --- > geometry
- **Result:** An sphere geometry which radius is equal to the operand made of 2 hemisphere.
- **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 <- slice(10,0.3);      // var0 equals a circle geometry of
radius 10, displayed as a slice.
```

- **See also:** [around](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#) , [hemisphere](#) , [pie3D](#) ,

[Top of the page](#)

solid

Same signification as [without_holes](#) operator.

[Top of the page](#)

sort

Same signification as [sort_by](#) operator.

[Top of the page](#)

sort_by

- Possible use:
 - container OP any expression --- > list
- **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:**

```
list var0 <- [1,2,4,3,5,7,6,8] sort_by (each);      // var0 equals
[1,2,3,4,5,6,7,8]
list var2 <- g2 sort_by (length(g2 out_edges_of each) );    // var2 equals
[node9, node7, node10, node8, node11, node6, node5, node4]
list var3 <- (list(node) sort_by (round(node(each).location.x)));    //
var3 equals [node5, node1, node0, node2, node3]
list var4 <- [1::2, 5::6, 3::4] sort_by (each);      // var4 equals [2, 4,
6]
```

- **See also:** [group_by](#),

[Top of the page](#)

source_of

- Possible use:
 - graph OP 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_specy":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](#),

[Top of the page](#)

spatial_graph

- Possible use:
 - OP(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](#),

[Top of the page](#)

species

- Possible use:
 - OP(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
```

[Top of the page](#)

species_of

Same signification as [species](#) operator.

[Top of the page](#)

sphere

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

spherical_pie

- Possible use:
 - float OP list<float> --- > geometry
 - float OP list<float> --- > geometry
- **Result:** An sphere geometry which radius is equal to the operand made of n pie.An sphere geometry which radius is equal to the operand made of n pie.
- **Comment:** the centre of the sphere is by default the location of the current agent in which has been called this operator.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.
 - returns a point if the operand is lower or equal to 0.
- **Examples:**

```
geometry var0 <- spherical_pie(10,[1.0,1.0,1.0]);      // var0 equals a
circle geometry of radius 10, displayed as a sphere with 4 slices.
geometry var1 <- spherical_pie(10/2,[0.1,0.9],[#red,#green]);    // var1
equals a circle geometry of radius 10, displayed as a sphere with 2 slices.
```

- **See also:** [around](#) , [cone](#) , [line](#) , [link](#) , [norm](#) , [point](#) , [polygon](#) , [polyline](#) , [rectangle](#) , [square](#) , [triangle](#) , [hemisphere](#) , [pie3D](#) ,

[Top of the page](#)

split_at

- Possible use:
 - geometry OP 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}}])]
```

[Top of the page](#)

split_geometry

- Possible use:
 - geometry OP float ---> list<geometry>
 - geometry OP point ---> list<geometry>
 - geometry OP 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 square cells of the given side size (geometry, size)A list of geometries that result from the decomposition of the geometry by rectangle cells of the given dimension (geometry, {size_x, size_y})
- **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_squares(self, 10.0);      // var1 equals the list
of the geometries corresponding to the decomposition of the geometry by
squares of side size 10.0
list<geometry> var2 <- to_rectangles(self, {10.0, 15.0});    // var2
equals the list of the geometries corresponding to the decomposition of the
geometry by rectangles of size 10.0, 15.0
```

[Top of the page](#)

—

split_lines

- Possible use:
 - OP(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}])
```

[Top of the page](#)

—

split_with

- Possible use:
 - string OP string --- > list
- **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:**

```
list 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']
```

[Top of the page](#)

—

sqrt

- Possible use:
 - OP(float) --- > float
 - OP(int) --- > 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
```

[Top of the page](#)

square

- Possible use:
 - OP(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](#)

[Top of the page](#)

standard_deviation

- Possible use:
 - OP(container) --- > float
- **Result:** the standard deviation on the elements of the operand. See < A href="http://en.wikipedia.org/wiki/Standard_deviation">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](#) ,

[Top of the page](#)

string

- Possible use:
 - OP(any) --- > string
- **Result:** Casts the operand into the type string

[Top of the page](#)

student_area

- Possible use:
 - float OP int --- > float

[Top of the page](#)

student_t_inverse

- Possible use:
 - float OP int --- > float

[Top of the page](#)

successors_of

- Possible use:

- graph OP unknown --- > list
- **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:**

```
list var1 <- graphEpidemio successors_of ({1,5});      // var1 equals
[12,45]
list var2 <- graphEpidemio successors_of node({34,56});    // var2 equals
[]
```

- **See also:** [predecessors_of](#), [neighbours_of](#),

[Top of the page](#)

sum

- Possible use:
 - OP(graph) --- > float
 - OP(container) --- > unknown
- **Result:** the sum of all the elements of the operand
- **Comment:** the sum operator 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 integer 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 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](#),

[Top of the page](#)

—

svg_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type svg. Allowed extensions are limited to svg

[Top of the page](#)

—

tan

- Possible use:
 - OP(float) --- > float
 - OP(int) --- > 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 expect infinity).
- **Examples:**

```
float var0 <- tan ( 0 );           // var0 equals 0.0
float var1 <- tan(90);           // var1 equals 1.633123935319537E16
```

- **See also:** [cos](#) , [sin](#) ,

[Top of the page](#)

—

tan_rad

- Possible use:
 - OP(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 expect infinity).
- **See also:** [cos](#) , [sin](#) ,

[Top of the page](#)

tanh

- Possible use:
 - OP(float) --- > float
 - OP(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
```

[Top of the page](#)

target_of

- Possible use:
 - graph OP 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_specy":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](#),

[Top of the page](#)

teapot

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

text_file

- Possible use:
 - OP(string) ---> file
- **Result:** Constructs a file of type text. Allowed extensions are limited to txt, data, csv, text, tsv, xml

[Top of the page](#)

threeds_file

- Possible use:
 - OP(string) ---> file
- **Result:** Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

[Top of the page](#)

threeds_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

[Top of the page](#)

to_gaml

- Possible use:
 - OP(unknown) --- > string
- **Result:** represents the gaml way to write an expression in gaml, depending on its type
- **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
```

- **See also:** [to_java](#),

[Top of the page](#)

to_rectangles

Same signification as [split_geometry](#) operator.

- Possible use:
 - geometry OP point --- > list<geometry>
 - geometry OP int --- > list<geometry>
- **Result:** 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 geometryA 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
- **Examples:**

```
list<geometry> var0 <- to_rectangles(self, 5, 20, true);      // var0 equals the list of rectangles corresponding to the discretisation 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
list<geometry> var1 <- to_rectangles(self, {10.0, 15.0}, true);    // var1 equals the list of rectangles of size {10.0, 15.0} corresponding to the discretisation into rectangles of the geometry of the agent applying the operator. The rectangles overlapping the border of the geometry are kept
```

[Top of the page](#)

to_squares

- Possible use:
 - geometry OP float --- > list<geometry>
- **Result:** 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
- **Examples:**

```
list<geometry> var0 <- to_squares(self, 10.0, true);      // var0 equals the list of squares of side size 10.0 corresponding to the discretisation into squares of the geometry of the agent applying the operator. The squares overlapping the border of the geometry are kept
```

[Top of the page](#)

to_triangles

Same signification as [triangulate](#) operator.

[Top of the page](#)

tokenize

Same signification as [split_with](#) operator.

[Top of the page](#)

topology

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

— touches

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

— towards

- Possible use:

- geometry OP 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](#) ,

[Top of the page](#)

transformed_by

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

translated_by

- Possible use:
 - geometry OP 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](#) ,

[Top of the page](#)

translated_to

Same signification as [at_location](#) operator.

[Top of the page](#)

triangle

- Possible use:
 - OP(float) ---> geometry
- **Result:** A triangle geometry which side size is given by the operand.
- **Comment:** the centre 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](#)

[Top of the page](#)

triangulate

- Possible use:
 - OP(list<geometry>) ---> list<geometry>
 - OP(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.
```

[Top of the page](#)

truncated_gauss

- Possible use:
 - OP(point) --- > float
 - OP(list) --- > 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 an float
between -0.3 and 0.3
float var1 <- truncated_gauss ([0.5, 0.0]);     // var1 equals 0.5
```

- **See also:** [gauss](#) ,

[Top of the page](#)

undirected

- Possible use:
 - OP(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](#) ,

[Top of the page](#)

union

- Possible use:
 - OP(container<geometry>) --- > geometry
 - container OP container --- > list
- **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
list var1 <- [1,2,3,4,5,6] union [2,4,9];      // var1 equals
[1,2,3,4,5,6,9]
list var2 <- [1,2,3,4,5,6] union [0,8];      // var2 equals
[1,2,3,4,5,6,0,8]
list 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](#) , [+](#) ,

[Top of the page](#)

unknown

- Possible use:
 - OP(any) --- > unknown
- **Result:** Casts the operand into the type unknown

[Top of the page](#)

use_cache

- Possible use:
 - graph OP 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](#),

[Top of the page](#)

user_input

- Possible use:
 - OP(any expression) --- > map<string,unknown>
 - string OP any expression --- > map<string,unknown>
- **Result:** asks the user for some values (not defined as parameters)
- **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> 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"))];
map<string,unknown> values <- user_input([ "Number" :: 100, "Location" :::
{10, 10} ]);
create bug number: int(values at "Number") with: [location::: (point(values
at "Location"))];
```

[Top of the page](#)

variance

- Possible use:
 - OP(container) --- > float
- **Result:** the variance of the elements of the operand. See < A href="<http://en.wikipedia.org/wiki/Variance>">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](#) ,

[Top of the page](#)

variance1

- Possible use:
 - OP(float) --- > float

[Top of the page](#)

variance2

- Possible use:
 - int OP float --- > float

[Top of the page](#)

voronoi

- Possible use:
 - OP(list<point>) --- > list<geometry>
- **Result:** A list of geometries corresponding to the Voronoi diagram built from the list of points
- **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.
```

[Top of the page](#)

water_area_for

- Possible use:
 - geometry OP float --- > float

- **Special cases:**

- if the left operand is a polyline and the right operand a float for the water y coordinate, returns the area of the water (water flow area)

- **Examples:**

```
waterarea <- my_river_polyline water_area_for my_height_value
```

[Top of the page](#)

water_level_for

- Possible use:

- geometry OP float --- > float

- **Special cases:**

- if the left operand is a polyline and the right operand a float for the area, returns the y coordinate of the water (water level)

- **Examples:**

```
waterlevel <- my_river_polyline water_level_for my_area_value
```

[Top of the page](#)

water_polylines_for

- Possible use:

- geometry OP float --- > list<list<point>>

- **Special cases:**

- if the left operand is a polyline and the right operand a float for the water y coordinate, returns the shapes of the river sections (list of list of points)

- **Examples:**

```
waterarea <- my_river_polyline water_area_for my_height_value
```

[Top of the page](#)

weight_of

- Possible use:
 - graph OP 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
```

[Top of the page](#)

where

- Possible use:
 - container OP any expression --- > list
- **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

```
list var4 <- [1::2, 3::4, 5::6] where (each >= 4);           // var4 equals [4,  
6]
```

- **Examples:**

```
list var0 <- [1,2,3,4,5,6,7,8] where (each > 3);           // var0 equals [4, 5,  
6, 7, 8]  
list var2 <- g2 where (length(g2 out_edges_of each) = 0 );      // var2  
equals [node9, node7, node10, node8, node11]  
list var3 <- (list(node) where (round(node(each).location.x) > 32)); //  
var3 equals [node2, node3]
```

- **See also:** [first_with](#) , [last_with](#) , [where](#) ,

[Top of the page](#)

with_max_of

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

with_min_of

- Possible use:
 - container OP 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](#) ,

[Top of the page](#)

with_optimizer_type

- Possible use:
 - graph OP 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](#) ,

[Top of the page](#)

with_precision

- Possible use:
 - point OP int --- > point
 - float OP int --- > float
- **Result:** Rounds off the ordinates of the left-hand point to the precision given by the value of right-hand operandRounds 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](#),

[Top of the page](#)

with_weights

- Possible use:
 - graph OP list ---> graph
 - graph OP 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)
```

[Top of the page](#)

without_holes

- Possible use:
 - OP(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.
```

[Top of the page](#)

writable

- Possible use:
 - file OP 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](#) ,

[Top of the page](#)

xml_file

- Possible use:
 - OP(string) --- > file
- **Result:** Constructs a file of type xml. Allowed extensions are limited to xml

[Top of the page](#)

7. Recipes

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 [G__UsingEquations 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 [G__UsingFIPAACL 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.

7.1 Optimizing Models

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`'.

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;
loop times: 1000 {
    bug one_big_bug <- one_of (bug where (each.size > 10));
}
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 : [ ]) :
[ ]
{
    ...
}
```

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). 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). Take a look at the diagrams in Issue668. Impressive, uh ? 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. Note that we don't even consider the **with_min_of** operator here):

```
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 syntax #2 above: 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 (see Issue 668 again), 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).

—

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;
}
```

7.4 Using Database Access

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.extensions.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 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
<i>dbtype</i>	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)
<i>host</i>	Yes	Host name or IP address of data server. It is absent when we work with SQLite.
<i>port</i>	Yes	Port of connection. It is not required when we work with SQLite.
<i>database</i>	No	Name of database. It is the file name including the path when we work with SQLite.
<i>user</i>	Yes	Username. It is not required when we work with SQLite.
<i>passwd</i>	Yes	Password. It is not required when we work with SQLite.
<i>srid</i>	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. ||

<i>port</i>	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) ||

<i>user</i>	No	Username.
<i>passwd</i>	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 Mondrian 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 Mondrian 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 [GamaRuntimeException].

- **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
<i>dbtype</i>	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)
<i>host</i>	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. ||

<i>database</i>	No	Name of database. It is the file name including the path when we work with SQLite.
<i>user</i>	Yes	Username. It is not required when we work with SQLite.
<i>passwd</i>	Yes	Password. It is not required when we work with SQLite.

<i>srid</i>	Yes	<i>srid</i> (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.
<i>select</i>	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];
    }
}

```

7.5 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.

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;
    }
}
```

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::[3.33333333333333]

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]

7.8 Defining User Interaction

Defining User Interaction

—
GAMA provides tools to define user interactions during the simulation to give more flexibility to the user in controlling the agents, creating agents, killing agents, running specific actions, etc:

- [Event](#) : allows to define specific user interaction actions for a display through a specific layer
- [User command](#) : allows to let the user call specific action through the user interface.
- [User control architecture](#) : allows to give the user the control on an agent.

Event

Events allow to interact with the simulation by capturing mouse events and do an action. This action could apply a change on the environment or on agents, according to the goal. An event is defined for a display. Several events can be defined for a same display. To define a event, the modelers has to add a event layer to a display:

```
display my_display {  
    event [event_type] action: myAction  
}
```

- event_type: **mouse_down** or **mouse_up**
- myAction is an action written in the global block. This action have to follow the specification below.

Specification of the action to define:

```
global  
{  
    ...  
    action myAction (point lot, list selected_agents)  
    {  
        // lot: contains the location of the click in the environment  
        // selected_agents: contains agents clicked by the event  
  
        ... //code written by the authors ...  
    }  
}
```

For example:

```
experiment Displays type: gui {
    output {
        display View_change_color {
            species cell;
            event [mouse_down] action: change_color;
        }
        display View_change_shape {
            species cell;
            event [mouse_down] action: change_shape;
        }
    }
}
```

User Command

Introduction to user commands

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::val1,
arg2::val2, ...];
```

or

```
user_command kill_myself {
    do die;
}
```

These commands (which belong to the "top-level" statements like actions, reflexes, etc.) are not executed when an agent runs. Instead, they are collected and used as follows:

- When defined in a GUI experiment, they appear as buttons above the parameters of the simulation;
- When defined in the global block or in any species,
 - when the agent is inspected, they appear as buttons above the agents' attributes
 - 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];
    }
}
```

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` operator

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, 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 like the following one to force the user to input new values instead of relying on the initial values of parameters :

```
global {
```

```

init {
    map values <- user_input([ "Number" :: 100]);
    create my_species number : int(values at "Number");
}

```

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 controled 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 (ie. 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.

References

References

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:

- Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, Alexis Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'The 16th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Dunedin, New Zealand, Volume 8291, pp. 242-258.

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).
- Arnaud Grignard, Patrick Taillandier, Benoit Gaudou, Duc An Vo, Nghi Quang Huynh, Alexis Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'The 16th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Dunedin, New Zealand, Volume 8291, pp. 242-258.
- 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 117-131.
- 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', Bangkok, 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

2014

- 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.